VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Rozšíření HDF5 o ukládání časových řad

Extension of HDF5 by Time Series Storing

2016 Martin Moudrý

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Martin Moudrý**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Rozšíření HDF5 o ukládání časových řad
Extension of HDF5 by Time Series Storing

Jazyk vypracování: čeština

Zásady pro vypracování:

Pro uložení velkých objemů dat jsou v dnešní době používány různé přístupy, mezi které patří například NoSQL databáze a také speciální datové formáty pro uložení dat, mezi které patří například HDF5. Tento hierarchický datový formát a jeho implementace nám umožňuje nejen ukládat velké objemy dat, ale je také optimalizován pro přístup z výpočetních uzlů HPC clusteru. V rámci této diplomové práce se student bude zabývat implementací nadstavby nad HDF5 knihovnou pro uložení časových řad, a to včetně analýzy možnosti komprese těchto dat. K implementované knihovně bude vytvořená dokumentace tak, aby bylo možno nad ní implementovat různé typy adaptéru pro její využití v technologiích, jako jsou C#, R, python a pod. Student se také zaměří na srovnání HDF5 s jinými typy uložení dat používanými v HPC prostředí (například Adios, Lustre).

Jednotlivé body zadání jsou:
1. Prozkoumat datový formát HDF5 a jeho možnosti ve vztahu k problematice ukládání časových řad.
2. Prozkoumat možnosti ukládání velkých dat pro HPC clustery.
3. Navrhnout a implementovat nadstavbu HDF5 pro práci s časovými řadami.
4. Popsat API k implementované knihovně.
5. Experimentálně ověřit funkčnost knihovny na testovací kolekci dat.

Seznam doporučené odborné literatury:

[1] HDF5 User's Guide: https://www.hdfgroup.org/HDF5/doc/UG/UG_frame10Datasets.html
[2] HDF5 Application Developer's Guide: https://www.hdfgroup.org/HDF5/doc/ADGuide.html

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jan Martinovič, Ph.D.**

Datum zadání: 01.09.2015

Datum odevzdání: 29.04.2016

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 29. dubna 2016

Moudrý

.....

Na tomto místě bych rád poděkoval Ing. J. Martinovičovi, Ph.D. za jeho cenné rady a připomínky při vedení mé diplomové práce.

Abstrakt

Tato diplomová práce se zabývá ukládáním velkých dat, převážně ukládáním velkých časových řad. V práci jsou zmíněny různé přístupy k jejich ukládání, zabývá se NoSQL databázemi, knihovnou Adios a vědeckými datovými formáty HDF5, CDF a NetCDF. Dále práce popisuje souborový systém Lustre a jeho vliv na paralelní přístup k datům. Poté se zabývá vytvořením nadstavby nad HDF5 knihovnou pro uložení časových řad a popsáním jejího použití. Následně je tato nadstavba otestována na výpočetních uzlech HPC.

Klíčová slova: Big data, Časové řady, NoSQL, Adios, HDF5, CDF, NetCDF, HPC

Abstract

This thesis deals with storing of big data, mostly with storing big time series. There are mentioned different approaches to their storing, NoSQL databases, Adios library and Scientific Data formats HDF5, CDF and NetCDF. After that, it describes Lustre file system and its influence on paralel data access. Then it deals with creating extension on HDF5 for storing time series data and describes usage of this extension. Finally this extension is tested on HPC compute nodes.

Key Words: Big data, Time series, NoSQL, Adios, HDF5, CDF, NetCDF, HPC

Obsah

Seznam použitých zkratek a symbolů

Adios	–	Adaptable IO System
API	–	Application Programming Interface
CDF	–	Common Data Format
Georgia Tech	–	Georgia Institute of Technology
HDF	–	Hierarchical Data Format
HPC	–	High-performance computing
I/O	–	Input/Output
JSON	–	JavaScript Object Notation
MDS	–	Metadata Server
MDT	–	Metadata Target
MGS	–	Management Server
MGT	–	Management Target
MPI	–	Message Passing Interface
NCSA	–	National Center for Supercomputing Applications
NetCDF	–	Network Common Data Form
NoSQL	–	Not only SQL
OpenSFS	–	Open Scalable File Systems
ORNL	–	Oak Ridge National Laboratory
OSS	–	Object Storage Server
OST	–	Object Storage Target
POSIX	–	Portable Operating System Interface
RAM	–	Random Access Memory
SQL	–	Structured Query Language
UCAR	–	University Corporation for Atmospheric Research
XML	–	Extensible Markup Language

Seznam obrázků

Seznam tabulek

1 Úvod

S velkým rozvojem počítačů a techniky v poslední době přibývá také velké množství dat. Některá z těchto nových dat jsou tak velká nebo tak komplexní, že je nelze zpracovávat běžně používanými prostředky v přijatelném čase. Těmto velkým datům se říká Big data (viz [3]). Musíme tak řešit problém ukládání a zpracování těchto dat. Vzhledem k velikosti dat je také často nutné pracovat s daty komprimovanými, ale to má zase negativní vliv na výkon. Pro řešení tohoto problému vznikají nové přístupy k ukládání dat, které nabízí efektivní přístup k těmto datům, často i v souvislosti s kompresí. Jednou z možností, kterou také máme je zpracovávat tyto data na HPC clusterech. I proto je důležité, aby řešení těchto problémů podporovalo paralelní přístup a dalo se tak efektivně použít na HPC clusterech, které se často využívají pro zpracování takhle velkých dat.

V této práci budou popsány různé přístupy k ukládání velkých dat, převážně ve formě časových řad. Kapitola 2 bude zaměřena na takzvané vědecké datové formáty, které byly speciálně vytvořeny pro ukládání těchto velkých dat. V kapitole 3 budou popsány tradičnější přístupy pro ukládání dat na HPC clusterech, konkrétně to budou databáze typu NoSQL a paralelní I/O knihovna Adios. V této kapitole bude také popsán souborový systém Lustre a jeho potenciální přínos při paralelním přístupu k datům. Následně bude v kapitole 4 podrobněji popsán vědecký datový formát HDF5.

V kapitole 5 bude popsána nástavba nad HDF5 pro ukládání časových řad, která v rámci této diplomové práce vznikla. A nakonec budou v kapitole 6 prezentovány testy, pomocí kterých byla tato nástavba otestována.

2 Vědecké datové formáty

Vzhledem k velikosti, jaké dosahují Big data, není možné tyto data efektivně ukládat v souborech typu csv nebo XML. Další možností jsou binární soubory, které ale postrádají popis uložených dat a nejsou proto vhodné pro jejich archivaci. Alternativou jsou takzvané vědecké datové formáty (viz [24]), které v mnohém rozšiřují binární soubory.

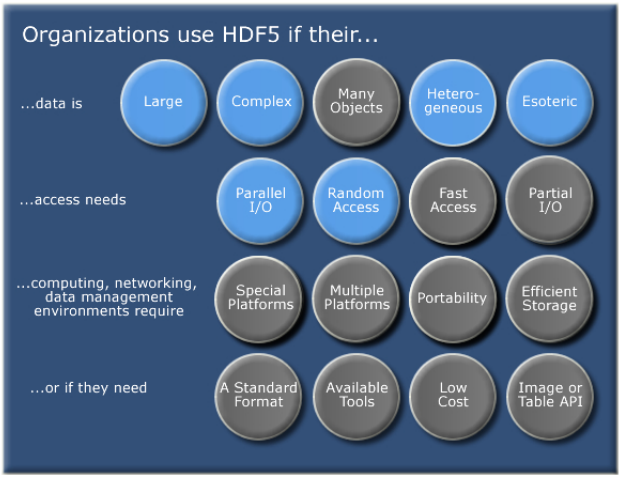
Vědecké datové formáty oproti binárním souborům obsahují popis ukládaných dat a jsou tedy vhodné pro jejich archivaci a přenositelnost mezi uživateli. Vědecké datové formáty také často řeší další problémy spojené s ukládáním velkých dat, jako jsou například přímý přístup nebo komprese.

Mezi nejvýznamnější vědecké datové formáty patří HDF (Hierarchical Data Format), CDF (Common Data Format) a NetCDF (Network Common Data Form), které v této kapitole popíšu.

2.1 HDF

HDF - Hierarchical Data Format (viz [27] a [14]) se začal vyvíjet v roce 1987 ve výzkumném centru NCSA univerzity Illinois (National Center for Supercomputing Applications - viz [19]). Od NCSA se v roce 2006 oddělila nezisková společnost The HDF Group, která hierarchický datový formát spravuje dodnes.

HDF je volně dostupný, unikátní soubor technologií sloužící pro správu extrémně velkých a komplexních dat i metadat. HDF je rovněž velmi flexibilní a umožňuje ukládání více různých datových objektů do jednoho souboru. HDF také nabízí rychlý a efektivní přístup k datům, umožňuje přímý přístup k datům a podporuje i paralelní přístup a kompresi dat. Jednou z výhod HDF je také to, že funguje na různých hardware architekturách i pod různými operačními systémy. V poslední řadě také HDF nabízí spoustu nástrojů, které můžeme použít při práci s HDF soubory. Důvody proč používat HDF dobře shrnuje obrázek 1.



Obrázek 1: Proč používat HDF (obrázek z [30])

Dalšími důležitými vlastnostmi HDF jsou rozšiřitelnost, hierarchičnost a samopopisnost souborů. Rozšiřitelnost nám umožňuje HDF soubory rozšiřovat i po jejich vytvoření a přidávat tak do nich další data. Hierarchičnost v tomto případě znamená, že uvnitř jednoho HDF souboru můžeme mít více různých datových objektů, které můžeme libovolně hierarchicky rozdělit na menší skupiny. Samopopisnost znamená, že každý HDF soubor obsahuje informace o tom jaké data obsahuje, jakého jsou tyto data formátu, dimenze, velikosti a různé další informace potřebné pro čtení těchto dat. To v praxi znamená, že pro přenos dat není třeba žádných dalších souborů, které bychom jinak museli mít v případě, že bychom používali nějaký vlastní binární formát.

HDF obsahuje nejen samotný datový formát, ale má i další vrstvy, které dohromady tvoří soubor nástrojů pro práci s HDF soubory. Tyto jednotlivé vrstvy můžeme vidět na obrázku 2.

Obrázek 2: Vrstvy HDF (obrázek z [28])

Na nejnižší úrovni leží samotný datový formát. Momentálně HDF obsahuje dva datové formáty, starší HDF4 a novější HDF5. Starší datový formát má spoustu problémů. Mezi největší problémy HDF4 patří jeho omezení na velikost souborů jen do 2 GB, chybějící komprese a paralelní přístup. I při sekvenčním přístupu je mnohem pomalejší než novější HDF5. I přes tyto problémy je stále společností The HDF Groups podporovaný, ale není už dále vyvíjený a nedoporučuje se jej používat pro vývoj nových aplikací. Veškerý vývoj se momentálně soustředí na formát HDF5, který byl speciálně vytvořen aby vyřešil nedostatky svého předchůdce. HDF5 se podrobněji budeme zabývat v kapitole 4.

V druhé vrstvě HDF se nachází low-level rozhraní v programovacím jazyce C, které slouží pro přímý přístup k datům, správu paměti a odchytávání chyb. Toto rozhraní je rovněž možné použít v případě, kdy je třeba naimplementovat nějaké funkce, které nejsou ve vyšších vrstvách HDF.

Třetí vrstvou HDF je aplikační programovací rozhraní, které slouží pro běžný přístup k

datům, vytváření a správu HDF souborů. Momentálně toto rozhraní podporuje programovací jazyky C, C++, Fortran 90 a Java. Toto API, stejně jako low-level rozhraní, je zaobalené do multiplatformní knihovny a mimo jiné je vhodné i pro masivně paralelní systémy.

Poslední vrstvou HDF jsou nástroje a aplikace pro správu, manipulaci, zobrazení a analýzu dat obsažených v HDF souborech. Významnou aplikací je HDFView, která slouží k zobrazování obsahu HDF souboru a všech informací, které se k tomuto souboru vážou. HDF také nabízí knihovnu H4toH5, která slouží pro převádění dat ze staršího HDF4 formátu na novější HDF5. Součástí HDF jsou i různé nástroje pro příkazovou řádku. Jedním z nich je h5repack, který udělá kopii HDF souboru s nějakou úpravou, typicky z nekomprimovaného souboru vytvoří komprimovaný. Dalším nástrojem je h5copy, který umožňuje zkopírovat datový objekt z jednoho souboru do jiného. Nástroj h5diff pak slouží k porovnání dvou HDF souborů a zobrazení jejich rozdílů. [1]

2.2 CDF

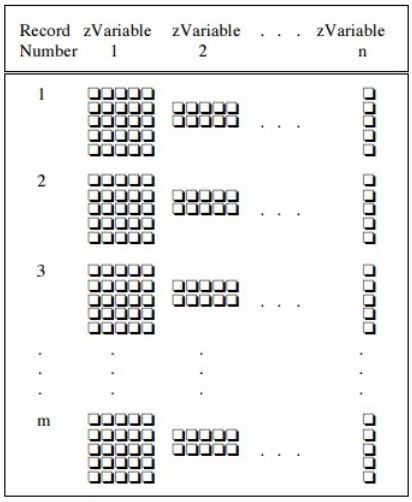
CDF - Common Data Format (viz [5] a [4]) je datový formát vyvíjený od roku 1985 v Goddardově kosmickém středisku NASA (viz [9]). Stejně jako u HDF se nejedná jen o datový formát, ale o sadu rozhraní, knihoven a aplikací sloužících pro práci se soubory uloženými v CDF formátu. CDF formát je také podporován velkým množstvím hardware architektur a operačních systémů. CDF nám také nabízí rozhraní v programovacích jazycích C, Java a Fortran.

Stejně jako HDF je i CDF samopopisný a rozšiřitelný datový formát. Podporuje tedy rozšiřování souborů a ve svých souborech obsahuje informace o datech, které obsahuje. Datový model CDF obsahuje dva typy objektů. Prvním typem objektů jsou takzvané Proměnné, které slouží pro ukládání samotných dat ve formě multidimenzionálních polí. CDF ještě rozlišuje mezi dvěma druhy těchto Proměnných a to mezi r-Proměnnými a z-Proměnnými. Rozdíl mezi těmito dvěma druhy je v tom, že pokud použijeme starší r-Proměnné, tak musí být v jednom CDF souboru všechny data stejné dimenze a velikosti těchto dimenzí musí být rovněž stejné. U novějších z-Proměnných už tento problém není, ale jejich nevýhodou je to, že se s nimi nedá pracovat přes standardní CDF rozhraní. U z-Proměnných se tedy musí použít vnitřní rozhraní, které je složitější. Tyto Proměnné se zapisují do pole, kterému se říká CDF-záznam. Často se toto pole používá pro reprezentaci času. Na obrázku 3 můžeme vidět jak může vypadat ukládání z-Proměnných.

Druhým typem objektů jsou takzvané Atributy, které reprezentují metadata. I tyto Atributy se dělí na dvě skupiny a to na g-Atributy což jsou globální metadata, tedy informace společné pro celý CDF soubor. Druhým typem jsou v-Atributy, které se vždy obsahují informace o jedné konkrétní Proměnné ke které se vážou.

CDF uživateli umožňuje ukládat data do dvou různých souborových formátů. Prvním formátem je single-file, ve kterém se všechny Proměnné a Atributy uloží do jednoho souboru. Druhým

[1] Celý seznam HDF aplikací a nástrojů nalezneme na: https://www.hdfgroup.org/products/hdf5_tools/index.html [datum poslední úpravy 24.3.2016]



Obrázek 3: Ukládání z-Proměnných v CDF (obrázek z [4])

formátem je pak multi-file, pomocí kterého se ukládají zvlášť všechny Atributy do jednoho souboru a každá Proměnná do svého vlastního souboru. Toto ukládání může mít výhodu v rychlosti přístupu, ale má také spoustu nevýhod a omezení. Vícesouborové ukládání nepodporuje kompresi, alokaci místa a nevýhodou je i práce s několika soubory najednou.

CDF podporuje přímý přístup k datům i kompresi v případě ukládání do jednoho souboru. Nevýhodou je, že velká část rozhraní nepodporuje novější z-Proměnné, což vede k složitějšímu používání CDF. Na rozdíl od HDF5 také CDF nepodporuje seskupování a hierarchii dat a je tedy omezenější co se týče typů dat, které může CDF reprezentovat.

2.3 netCDF

Posledním vědeckým formátem, kterým se budeme zabývat je NetCDF - Network Common Data Format (viz [21] a [20]). Tento formát byl původně vytvořený v roce 1989 korporací univerzit pro atmosférický výzkum UCAR (University Corporation for Atmospheric Research - viz [29]) na základě formátu CDF. Poté se ale oba formáty vyvíjely samostatně a v dnešní době už se v mnohém liší.

Stejně jako ostatní vědecké formáty je i NetCDF samopopisný, rozšiřitelný formát a spolu s formátem obsahuje i sadu rozhraní a knihoven mimo jiné pro jazyky C, C++, Fortran a Java. Rovněž i NetCDF je široce dostupné pod různými operačními systémy a hardware architekturami. NetCDF v dnešní době obsahuje celkem čtyři datové formáty, ze kterých si může uživatel vybrat.

Klasický formát je původní NetCDF formát vycházející z CDF. Je to defaultní NetCDF formát přesto má mnoho limitů. Mezi jeho limity patří chybějící komprese a omezená velikost dat. Proměnné klasického formátu nemůžou zabírat více než 2 GB místa. Druhým formátem který NetCDF obsahuje je 64-bit offset formát, který se od klasického liší jen v tom, že podporuje

větší velikosti souborů. I tak má, ale omezení na velikosti jednotlivých záznamů, proměnné musí zabírat méně než 4 GB místa.

V roce 2008 vznikl nový formát netCDF-4, který využívá pro své ukládání formát HDF5, nad kterým netCDF implementuje své rozhraní. Díky tomuto netCDF-4 podporuje kompresi, komplexnější datové typy a vyšší výkon než starší formáty. Přesto netCDF-4 nevyužívá všechny možnosti formátu HDF5 jako jsou například cykly v hierarchické struktuře nebo odkazy na objekty a data uvnitř HDF5 souboru. Díky tomu nelze k souborům vytvořeným pomocí HDF5 přistupovat přes NetCDF, je ale možné k souborům vytvořeným pomocí NetCDF přistupovat přes HDF5.

Posledním formátem je netCDF-4 klasický formát, který stejně jako netCDF-4 formát používá pro své ukládání formát HDF5. Stejně tak podporuje kompresi a využívá vyššího výkonu HDF5 souborů. Narozdíl od netCDF-4 formátu, ale nevyužívá komplexnosti HDF5, díky čemuž může používat původní interface.

Výhodou netCDF je, že obsahuje většinu funkčnosti HDF5 za přínosu svého jednoduššího rozhraní. Formát netCDF-4, ale nevyužívá všech možností HDF5 a je tak v některých ohledech oproti formátu HDF5 slabší. Nevýhodou také je, že v případě budoucího vývoje HDF5 bude netCDF se svým formátem netCDF-4 ve vývoji vždy pomalejší.

2.4 Srovnání

Při porovnávání těchto 3 formátů jsme čerpali z dokumentací jednotlivých formátů a z práce An Introduction to Distributed Visualization, kterou napsal Jan Heijmans (viz [2, kapitola 4.2]). Všechny tři vědecké formáty, které jsem v této kapitole popsal jsou samopopisné, rozšiřitelné, multiplatformní formáty, které podporují přímý přístup k datům a kompresi. Zároveň ke každému formátu existují rozhraní a knihovny pro práci s těmito formáty v mnoha programovacích jazycích. Nejslabší z těchto formátů se zdá být CDF, které nepodporuje seskupování dat a má poněkud starší a hůře použitelné rozhraní. Oproti tomu NetCDF má výhodu nejjednoduššího a nejsnáze použitelného rozhraní ze všech tří formátů. NetCDF také podporuje seskupování dat, jeho nevýhodou je ale závislost na HDF5 formátu, nad kterým staví. Hierarchický datový formát je ze všech tří formátů nejsilnější, obsahuje teoreticky neomezené množství datových typů, hierarchii objektů. Nevýhodou oproti NetCDF je, že má složitější rozhraní.

3 Ukládání dat na HPC clusterech

Vědecké datové formáty nejsou jedinou možností jak ukládat velká data. Tradičnějším způsobem ukládání je použití databázových systémů. V souvislosti s HPC se pak často využívají databáze typu NoSQL (viz [22]), které budou popsány v první části této kapitoly.

Pokud se zaměříme přímo na HPC a nebudeme trvat na multiplatformnosti, tak se nabízí ještě další způsoby pro ukládání dat. Jedním z těchto přístupů je použití paralelní I/O knihovny Adios, kterými se budeme zabývat ve druhé části.

Ukládání dat na HPC, ale neovlivňuje jen to jestli použijeme vědecké formáty, databázi nebo paralelní knihovny. Můžeme ho také optimalizovat pomocí souborových systémů, které fungují na pozadí. Poslední část tedy bude zaměřena na souborový systém Lustre a jeho možné použití s ostatními technologiemi pro optimalizaci přístupu k datům na HPC.

3.1 NoSQL

Relační databáze jsou databáze ve kterých jsou data uložená ve formě tabulek (relací). Řádky tabulky reprezentují jednotlivé instance dat a mají svůj unikátní klíč pomocí kterého se k nim přistupuje. Sloupce tabulky pak reprezentují jednotlivé hodnoty atributů těchto instancí. Pro udržování a přístup do relačních databází se nejčastěji využívá programovací jazyk SQL.

Oproti relačním databázím existují tzv. nerelační databáze, kterým se také říká NoSQL (Not only SQL). V této části, která bude zaměřena na tyto NoSQL a jejich využití v HPC, jsme vycházeli převážně z publikací RDBMS to NoSQL: Reviewing Some Next-Generation Non-Relational Database's (viz [16]) a RDBMS vs NoSQL: Performance and Scaling Comparison (viz [23]).

NoSQL opoti relačním databázím nemají žádnou přesně danou strukturu pro ukládání dat. Existuje proto několik různých typů NoSQL databází na které je můžeme rozdělit. Čtyři nejvýznamnější z nich budou v této kapitole popsány. Jejich vizualizované datové modely společně s porovnáním s relačními SQL databázemi můžeme vidět na obrázku 4.

Prvním typem jsou databáze založené na ukládání klíče a hodnoty. V těchto databázích se jak už z názvu vyplývá ukládá dvojice klíč a hodnota. Podle klíče, který je pro každou hodnotu unikátní se pak přistupuje k jednotlivým hodnotám. Příkladem takové databáze jsou DynamoDB[2] a Oracle NOSQL Database[3].

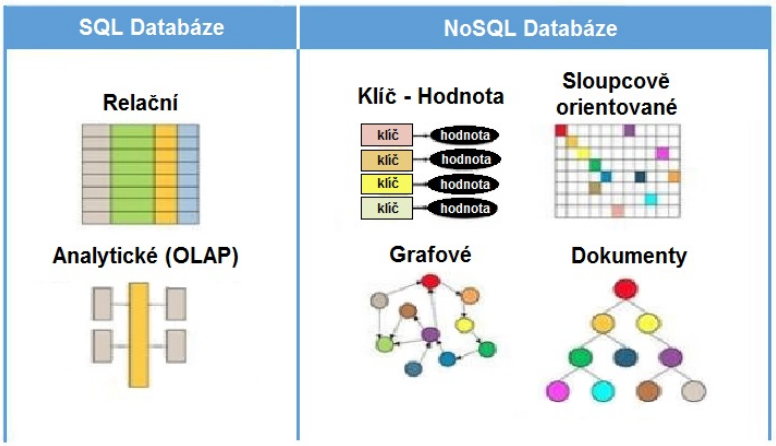
Druhým typem NoSQL databází jsou sloupcově orientované databáze. Rozdíl mezi těmito databázemi a relačními databázemi je v tom, že v těchto databázích se všechna data ukládají do jednoho sloupce tak, aby jednotlivé spolu související hodnoty byly hned za sebou. Příkladem sloupcově orientovaných databází jsou databáze Cassandra[4]. a HBase[5].

[2] http://aws.amazon.com/documentation/dynamodb/ [citováno 24.4.2016]

[3] http://www.oracle.com/technetwork/database/database-technologies/nosqldb/overview/index.html [citováno 24.4.2016]

[4] http://cassandra.apache.org/ [citováno 24.4.2016]

[5] https://hbase.apache.org/ [datum poslední úpravy 19.4.2016]



Obrázek 4: Srovnání SQL a různých typů NoSQL databází (obrázek z [7])

Třetím typem jsou databáze založené na ukládání dokumentů. V těchto databázích jsou data uložená ve formě kolekce dokumentů. Každý dokument reprezentuje nějaký datový objekt a může v něm být libovolný počet polí libovolné délky. Pro dokumenty se často používá notace JSON. Mezi tyto databáze patří mimo jiné MongoDB[6] a CouchDB[7].

Čtvrtým typem NoSQL jsou grafové databáze. V těchto databázích jsou data ukládána pomocí uzlů a hran. Uzly reprezentují danou instanci dat, zatímco hrany udávají, které uzly jsou spolu v relaci, tedy mají společnou nějakou ukládanou vlastnost. Každý uzel v sobě obsahuje odkaz na uzly, se kterými sousedí. Mezi tyto databáze patří například Neo4J[8] a Infinite Graph[9].

Vzhledem k tomuto velkému počtu různých typů databází a ještě většímu množství jednotlivých databází se velmi těžko generalizují vlastnosti těchto databází. Častými výhodami těchto databází jsou například vyšší výkon oproti relačním databázím a dobrá škálovatelnost. Toto je zejména důležité pro použití na HPC clusterech. NoSQL databáze jsou navíc velmi flexibilní a umožňují ukládat velké množství různých typů dat, jak strukturovaných, tak i nestrukturovaných. Díky velkému množství různých databází také často můžeme najít databázi, která je vhodná přímo pro naše data.

Častými nevýhodami NoSQL databází bývá například to, že nově uložená data na jednom NoSQL serveru nemusí být ihned viditelná na jiném, což může být pro některé aplikace nepřijatelné. Nevýhodou je také to, že pokud chceme NoSQL využívat ideálně, musíme pro každý typ dat pracovat s jinou NoSQL databází, která je pro tyto data určená.

Pokud srovnáme NoSQL databáze s HDF5 formátem s ohledem na velká data a HPC zjistíme, že mají tyto dva přístupy mnoho společného. Oba přístupy jsou velmi flexibilní a dokážou ukládat velké množství různých typů dat. Stejně tak oba přístupy dokážou zajistit vysoký výkon a jsou vhodné pro použití na HPC clusterech. Výhodou NoSQL databází oproti HDF5 je mož-

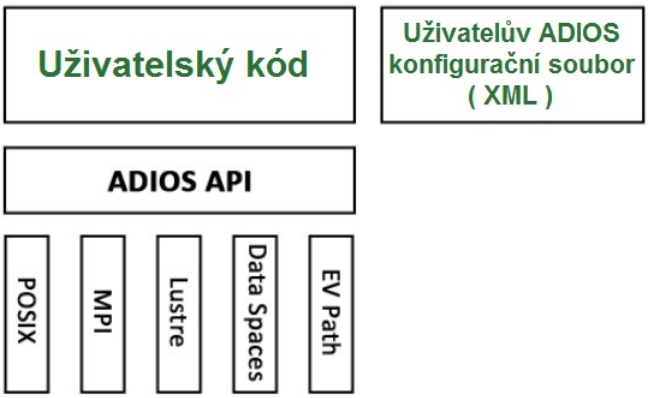
[6] https://www.mongodb.org/ [citováno 24.4.2016]
[7] http://couchdb.apache.org/ [citováno 24.4.2016]
[8] http://neo4j.com/ [citováno 24.4.2016]
[9] http://www.objectivity.com/products/infinitegraph/ [citováno 24.4.2016]

nost paralelního zápisu více uživatelů. Tuto funkci HDF5 postrádá. Další výhodou můžou být speciální vyhledávací dotazy, které některé NoSQL databáze můžou nabízet. HDF5 má oproti NoSQL výhodu v tom, že data všech různých typů můžeme efektivně ukládat do jednoho souboru, také se tyto data díky své velikosti a kompresi dají snáze přenášet a nezabírají tolik místa. Výhodou také je, že nemusíme hledat pro své data ideální databázi a testovat všechny možné NoSQL databáze.

V této práci se budu zabývat především HDF5, nicméně NoSQL databáze mohou být tomuto formátu dobrou alternativou. Je třeba ale zvolit vhodnou databázi z široké nabídky a pečlivě zvážit veškeré plusy a mínusy, které má každá tato databáze jiné. Záležet pak také bude na velikosti dat, práci s těmito daty a konkrétní architektuře na které bude spuštěna databáze. K jednoznačnému rozhodnutí, které z těchto dvou řešení je lepší, by tedy bylo zapotřebí rozsáhlé testování.

3.2 Adios

Adios - Adaptabilní I/O systém (viz [1] a [17]) vznikl jako projekt národní laboratoře Oak Ridge (ORNL) a institutu technologie v Goergii (Georgia Tech). Adios je knihovna sloužící k zjednodušení a urychlení paralelního I/O na HPC clusterech. Toho dosahuje tím, že dokáže zaměnit volání Adios funkcí za volání některé z podporovaných nižších vrstev. Mezi tyto podporované nižší vrstvy patří například POSIX, MPI-IO, DataSpaces, EVPath a Lustre. Model fungování Adiosu můžeme vidět na obrázku 5.



Obrázek 5: Model fungování knihovny Adios (obrázek z [17])

Ovládání Adiosu funguje pomocí konfiguračního XML souboru, do kterého se zapíšou informace o konkrétních datech a jak s nimi naložit. Díky tomuto je Adios velmi flexibilní a snadno přenositelný. Na nové platformě stačí jen zaměnit konfigurační soubor, ve kterém nastavíme vhodné I/O pro danou platformu. Adios také v konfiguračních souborech podporuje seskupování dat. Díky tomu je možné pro různé skupiny dat použít rozdílné I/O rozhraní.

Adios nabízí API v programovacích jazycích C a Fortran. Toto API je velmi univerzální a jednoduché a to právě díky tomu, že velká část funkcionality je určená až na základě konfiguračního souboru.

Pro ukládání dat a metadat používá Adios vlastní, samopopisný formát BP, který byl speciálně navržen pro HPC. Vlastnosti tohoto formátu jsou odolnost vůči náhlým výpadkům a takzvaná opožděná konzistentnost. To znamená že při zápisu více uživatelů najednou je tento formát odolnější vůči chybám způsobeným současným zápisem, ale za cenu krátkého zpoždění operací.

Porovnáme-li Adios s formátem HDF5 zjistíme, že mají tyto produkty mnoho společného. Oba slouží pro ukládání velkých dat s podporou paralelizace. Také se v obou případech jedná o nástavbu nad vlastním samopopisným datovým formátem. Adios i HDF5 také nabízí vysoký výkon a jsou vhodné pro HPC.

Výhodami Adiosu oproti HDF5 jsou možnost zvolení vhodného I/O rozhraní a podpora paralelního zápisu více uživatelů. Adios také oproti HDF5 nabízí jednodušší API. Nevýhodou Adiosu, ale může být, že na rozdíl od HDF5 není multiplatformní. Také je jeho API dostupné jen ve dvou programovacích jazycích (C a Fortran), což může být poněkud limitující.

Pro použití v HPC a pro velká data je knihovna Adios určitě vhodným formátem. Zvlášť pokud nepotřebujeme multiplatformnost HDF5 a dokážeme využít různé I/O rozhraní, které Adios nabízí. V manuálu knihovny Adios se mluví až o 1000 krát lepších výsledcích oproti jiným paralelním formátům, kterých je možné díky této knihovně dosáhnout. Tato práce se sice zabývá HDF5, ale pro další výzkum a pokračování v této oblasti by mohlo být velmi užitečné dopodrobna otestovat i knihovnu Adios.

3.3 Lustre

Do teď jsme se zabývali různými přístupy, které můžeme použít pro ukládání velkých dat na HPC clusterech. To jaký přístup zvolíme ale není jediná možnost jak optimalizovat přístup k datům. V této části bude popsán souborový systém Lustre, který dále optimalizuje přístup k datům pomocí fyzického rozdělení datových souborů na menší části a jejich uložením v rámci HPC clusteru.

Souborový systém Lustre vznikl jako výzkumný projekt Petra Braama v roce 1999. Od té doby ho vyvíjelo mnoho společností a momentálně se o jeho vývoj stará nezisková organizace OpenSFS. V této části vycházíme ze zdrojů [17] a [18].

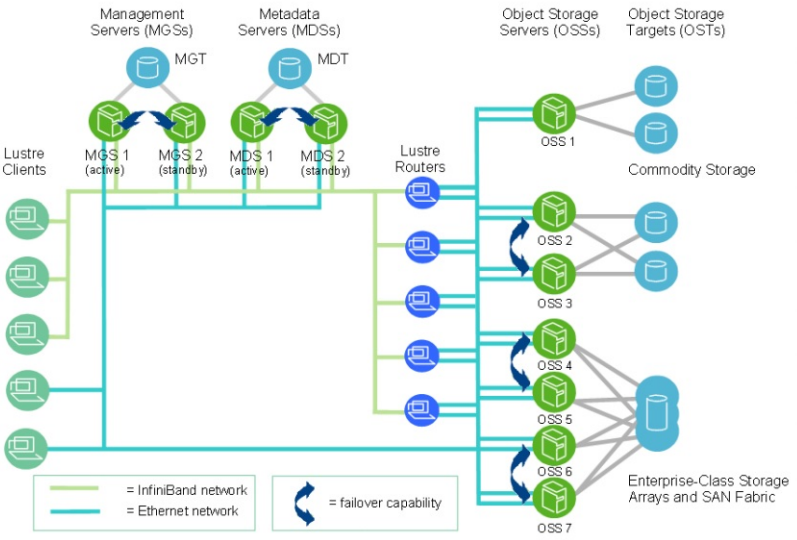
Architektura Lustre obsahuje několik typů objektů. Konfigurační informace souborového systému Lustre se ukládají do takzvaných MGT (Management Target), ke kterým je přistupováno pomocí MGS (Management Server). Tyto informace slouží pro fungování ostatních Lustre objektů. Každý souborový systém Lustre má jeden MGT a alespoň jeden MGS. Další MGS můžou sloužit jako záložní v případě výpadku.

Dalším typem objektů jsou MDT (Metadata Target), což jsou objekty, které slouží k ukládání metadat souborového systému jako jsou například názvy souborů, názvy složek nebo přístupová

práva. Přístup k těmto MDT pak spravují MDS (Metadata Server). Většinou má Lustre jeden MDT a alespoň jeden MDS, s tím, že další MDT a MDS zase slouží jako záložní. Je také možné spojit tyto metadata společně s konfiguračními informacemi, ale tento přístup se nedoporučuje, protože nedovoluje jejich nezávislou správu.

Obdobné to je i u samotného ukládání dat. Data se ukládají do objektů OST (Object Storage Target), ke kterým se přistupuje pomocí OSS (Object Storage Servers). Každý OSS pak většinou spravuje 2 až 8 OST a stejně jako u metadat je možné pro stejnou skupinu dat použít více těchto OSS, což slouží jak pro urychlení přístupu v případě více souběžných přístupů ke stejným datům, tak i jako záloha pro případ výpadku jednoho z OSS.

Posledním typem objektů jsou Lustre klienti, kteří slouží ke komunikaci mezi souborovým systémem a samotnými aplikacemi, které k němu přistupují. Většinou bývá jeden klient pro každou skupinku nodů na clusteru. Příklad celé architektury Lustre můžeme vidět na obrázku 6.



Obrázek 6: Architektura Lustre (obrázek z Manuálu k Lustre)

Princip Lustre spočívá v tom, že se každý ukládaný soubor rozdělí na pruhy dat a pak se každý tento pruh uloží do jiného OST. Tyto OST jsou pak fyzicky uložené v rozdílných částech HPC clusteru. Díky tomu paralelní proces přistupující k tomuto souboru v každém svém vlákně přistupuje do jiné části HPC clusteru, což optimalizuje propustnost mezi jednotlivými uzly. Uživatel má také možnost zvolit pro každý ukládaný soubor vlastní rozdělení souboru.

Souborový systém Lustre je použitelný pro mnoho různých typů clusterů, ale nejlepších výsledků dosahuje na HPC clusterech. Je totiž velmi dobře škálovatelný, do systému se dají dynamicky přidávat další OST i OSS což dále zvyšuje kapacitu, výkon i propustnost sítě. Lustre ale není vhodný pro případy kdy klient i servery běží na jediném uzlu a sdílejí stejný datový prostor. Důvod je ten, že v tomto případě chybí záložní mechanismy systému Lustre a v případě selhání budou data nedostupná až do restartování tohoto uzlu.

Pro optimalizaci přístupu k datům, je souborový systém Lustre velmi užitečným nástrojem. Díky tomu, že dokáže optimalizovat propustnost sítě, tak skvěle doplňuje paralelní přístupy k datům jako jsou HDF5 nebo Adios. Bez tohoto systému by totiž výkon těchto paralelních přístupů byl omezen nízkou propustností sítě, zatímco bez těchto paralelních přístupů by se neprojevila výhoda v rozdělení souborů mezi jednotlivé uzly.

Spojení HDF5 a Lustre je obzvlášť výhodné díky možnosti HDF5 rozdělit data do takzvaných chunků a pak v rámci I/O přistupovat ke každému chunku zvlášť. Toto se velmi hodí do konceptu Lustre, který tento soubor rozděluje i fyzicky a je tedy možné pomocí ideálního rozdělení HDF5 chunků a Lustre pruhů ještě lépe optimalizovat přístup k datům. Fungováním chunků a obecně celým HDF5 se budeme dále zabývat v kapitole 4.

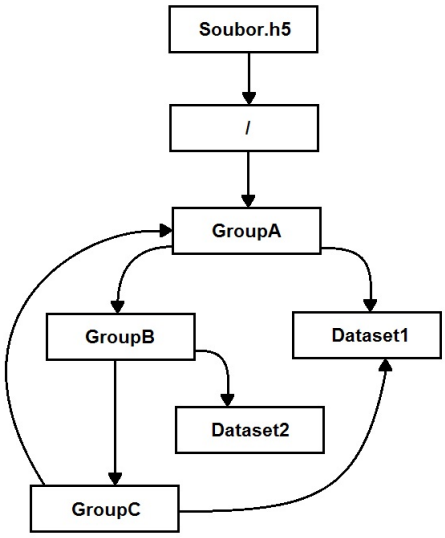
4 HDF5

V této kapitole se budeme dopodrobna zabývat formátem HDF5, který byl krátce zmíněn v kapitole 2. Vycházet budeme především z dokumentace k HDF5 (viz [11]). HDF5 je novější ze dvou hierarchických datových formátů, které vyvíjí společnost The HDF Group. Jak už bylo zmíněno, tak se jedná o volně dostupnou, multiplatformní knihovnu s API v programovacích jazycích C, C++, Fortran a Java. Součástí HDF5 je také vlastní datový formát pro ukládání dat. Tento datový formát je samopopisný, takže v sobě kromě dat ukládá i informace nutné pro zpětnou reprezentaci těchto dat.

4.1 Datový model formátu HDF5

HDF5 obsahuje univerzální datový model, který může reprezentovat velmi komplexní datové objekty. Tohoto dosahuje pomocí tří typů objektů, které každý HDF5 soubor může obsahovat.

Prvním typem objektů jsou takzvané groupy, které slouží k seskupování a hierarchii datových objektů. Jejich funkci můžeme přirovnat ke složkám souborového systému, který se tak uvnitř HDF5 souboru virtuálně vytváří. Každý soubor má jednu kořenovou groupu, ve které celý systém začíná. Tato kořenová groupa pak může obsahovat další groupy nebo jiné objekty. Tyto nové nové groupy pak znovu můžou obsahovat další objekty a celý systém díky tomu vytváří svou hierarchickou strukturu. Je také možné v této struktuře vytvářet různé cykly díky tomu, že do group můžeme přiřadit i již existující objekty z jiných group. Příklad jak může tento vnitřní souborový systém vypadat můžeme vidět na obrázku 7.



Obrázek 7: Struktura HDF5 souboru (obrázek z dokumentace k HDF5)

K jednotlivým objektům v hierarchické struktuře přistupujeme stejným způsobem jako u cest v unixových systémech. Kořenová groupa se značí pomocí „/“. Cestu ke groupě GroupB z

obrázku 7 můžeme zadat například tímto způsobem „/GroupA/GroupB/“, ale díky cyklu v dané struktuře můžeme použít mimo jiné i toto „/GroupA/GroupB/GroupC/GroupA/GroupB/“.

Druhým typem objektů jsou datasety, které slouží k samotnému ukládání dat. Jedná se o multidimenzionální pole určitého datového typu. Jejich dimenze, stejně jako velikosti těchto dimenzí jsou teoreticky neomezené. HDF5 také podporuje rozšiřování jednotlivých dimenzí s přibývajícími daty.

Celý dataset musí mít jen jeden stejný datový typ, ale různé datasety v rámci jednoho HDF5 souboru můžou mít různé datové typy. Základními datovými typy, které HDF5 podporuje jsou atomické typy jako jsou float, double, hhar nebo integery různé délky, ale i složené typy jako jsou pole, pole různé délky, výčtové datové typy a struktury. HDF5 obsahuje také různé možnosti pro úpravu těchto datových typů. Například je možné vytvořit integer specifické bitové délky. Je tak teoreticky možné dosáhnout jakéhokoliv datového typu.

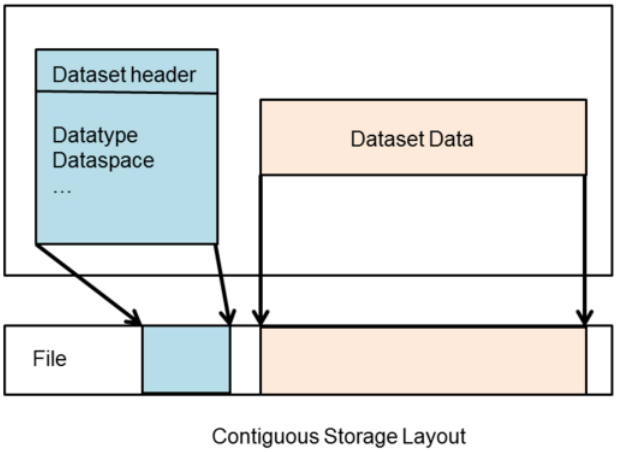
Jedním z problémů multiplatformních aplikací je to, že na mnoha platformách jsou datové typy v paměti realizovány různě. Na některých platformách může integer zabírat v paměti 4 byty, na jiných to může být 8 bytů. Tento problém v HDF5 řeší takzvané přirozené datové typy, které fungují jako aliasy pro reálné datové typy. Pokud je použijeme během čtení, tak se datový typ dat uložených v souboru převede na datový typ dat v paměti. To samé, ale naopak probíhá během zápisu. Díky tomu se velmi zjednodušuje práce uživatelů využívajících stejná data na různých platformách.

Posledním typem objektů jsou atributy. Atributy jsou malé datasety sloužící pro ukládání metadat a dodatečných informací. Atributy se vždy vážou k jednomu konkrétnímu objektu, buď ke groupě nebo k datasetu. Každý objekt HDF5 může obsahovat neomezený počet těchto atributů. Atributy můžou obsahovat stejné datové typy jako datasety. Na rozdíl od nich, ale neumožňují chunkování, kompresi ani částečný přístup. Při čtení a zápisu atributů se vždy musí přečíst celý atribut. Proto se hodí jen pro velmi malé objekty a v případě, že je potřeba uložit větší množství metadat, je lepší k tomuto účelu použít datasety.

4.2 Možnosti pro ukládání dat

V předchozí části byly mimo jiné popsány datasety, což jsou objekty HDF5 určené pro ukládání samotných dat. V této části se budeme zabývat různými možnostmi, které nám HDF5 formát pro ukládání datasetů nabízí. Pro ukládání datasetů můžeme zvolit ze tří základních typů uložení.

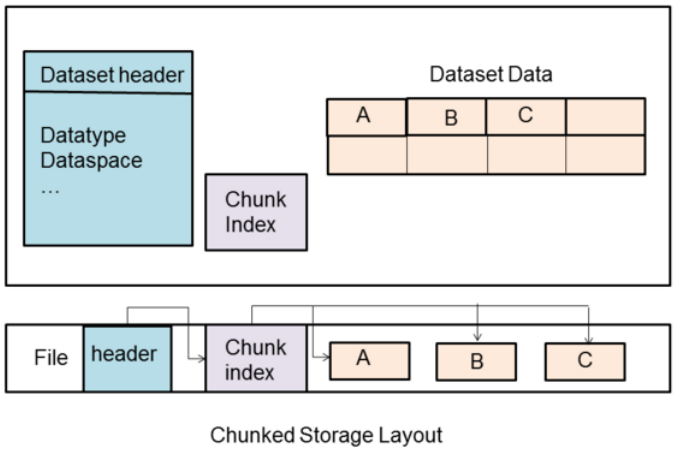
Prvním a defaultním typem ukládání je kontinuální typ. Při použití tohoto typu se data ukládají na disk jako jedno velké souvislé pole bitů. Velikost kontinuálního datasetu je třeba nastavit hned při vytvoření a proto HDF5 nepodporuje rozšiřování těchto datasetů. Další nevýhody kontinuálního typu ukládání jsou ty, že neumožňuje chunkování ani kompresi dat. Tento typ ukládání je vhodný pro středně velká data, která nebudou přibývat dynamicky a nepotřebujeme u nich kompresi ani výhody spojené s chunkováním. Příklad kontinuálního typu ukládání můžeme vidět na obrázku 8.



Obrázek 8: Kontinuální typ ukládání (obrázek z tutoriálu HDF5)

Druhým typem ukládání je ukládání kompaktní. Tento typ ukládání je vhodný jen pro velmi malé datasety, protože se dataset ukládá rovnou v hlavičce datasetu. Výhodou je, že se při čtení může hlavička i data těchto datasetů přečíst v jediné operaci. Kompaktní typ nepodporuje chunkování, kompresi, ale ani náhodný přístup. Svými vlastnostmi se tak podobá spíše atributům.

Posledním a pro velká data nejzajímavějším typem ukládání je ukládání po chuncích. Celý dataset je rozdělený do chunků, které jsou uloženy každý zvlášť kontinuálním způsobem. Ukládání po chuncích podporuje i kompresi a společně s výhodami spojenými se samotným rozdělením datasetu se tak velmi hodí pro velká data a paralelní přístup. Příklad ukládání po chuncích můžeme vidět na obrázku 9. O chunkování se více dozvíme v následující části.



Obrázek 9: Typ ukládání po chuncích (obrázek z tutoriálu HDF5)

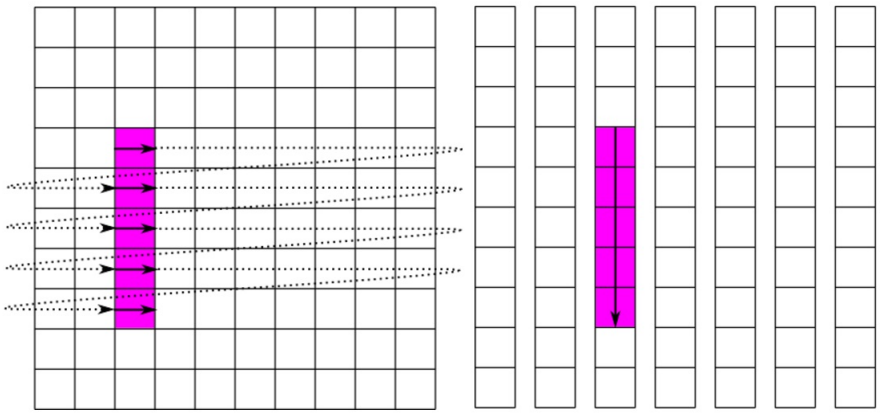
4.2.1 Chunkování

Chunkování v HDF5 je proces, kdy se datasety rozdělí na menší celky a ty se pak ukládají zvlášť. Chunky jsou nezbytným předpokladem pokud chceme využívat rozšiřitelnost datasetů

nebo kompresi. V celém datasetu musí mít chunky jednotnou velikost, nicméně tuto velikost je možné libovolně zvolit.

Právě volba správné velikosti chunků je velmi důležitá pro optimalizování výkonu i velikosti HDF5 souborů. Problémem je, že tuto ideální velikost není jednoduché určit. Existuje totiž jen několik doporučení, kterých by se měl uživatel držet při volbě velikosti chunků.

Prvním doporučením je, že by velikosti chunků měly co nejblíže odpovídat velikostem operací, které se budou nejčastěji nad daným datasetem provádět. Splnění tohoto doporučení má významný vliv na celkový výkon během I/O a to díky tomu, že stačí během takovéto operace načíst jen jediný chunk a v ideálním případě pak v jediném volání přečíst veškerá data, která tento chunk obsahuje. Ilustraci toho jak může dobře zvolená velikost chunků vylepšit výkon během I/O můžeme vidět na obrázku 10.

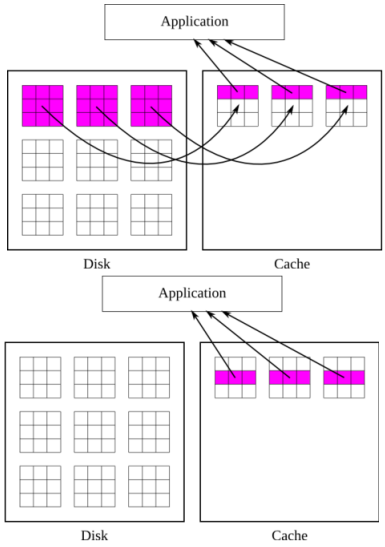


Obrázek 10: Výhody chunkování při čtení, vlevo kontinuální typ, vpravo ukládání po chuncích (obrázek z dokumentace k HDF5)

Velikost chunků by také neměla být moc malá, protože se zvyšujícím se počtem chunků se také zvyšuje čas na nalezení správného chunku během I/O operací. Malé velikosti chunků také vedou k velkému množství těchto operací. Chunky by, ale neměly být ani příliš velké, protože během I/O se načítá celý chunk a velké velikosti chunků by mohly výrazně zpomalit výkon. Velká velikost chunků je také problematická v souvislosti s kompresí. Během I/O operací nad komprimovaným datasetem se totiž nejdříve musí celý chunk dekomprimovat, následně se provede operace a nakonec se chunk znovu zkomprimuje na disk. Pro velké velikosti chunků by toto bylo velmi výrazným zpomalením.

Výkon není jediná věc, kterou velikost chunků ovlivňuje. Velikost chunků může také mít vliv na celkovou velikost souborů. HDF5 totiž alokuje vždy celý chunk, pokud alespoň jedno pole tohoto chunku obsahuje data. Toto může mít vliv na okrajích datasetu, kde může být z celého alokovaného chunku využita jen část. Zároveň to může mít vliv i v případě, že máme data s nějakými výpadky. Pokud tedy očekáváme pravidelné výpadky určité velikosti a nebo neočekáváme rozšiřování datasetu v některé z dimenzí, můžeme správným nastavením velikostí chunků optimalizovat i celkovou velikost souborů.

Samotné rozdělení datasetu není to jediné co nám chunkování nabízí. Další výhodou je možnost využití chunk cache. Chunk cache je paměť, kterou si každý dataset vyhradí pro načítání chunků. Díky tomu je možné mít zároveň načtené všechny chunky na kterých zrovna provádíme I/O a nemusíme je načítat během jedné operace vícekrát. Příklad takového využití chunk cache můžeme vidět na obrázku 11. Také u chunk cache můžeme zvolit její velikost a významně tak ovlivnit výkon HDF5. Chunk cache by měla být dostatečně velká na to, aby dokázala uložit alespoň tolik chunků, z kolika jich chceme číst během jediné operace. Zároveň je třeba si uvědomit, že čím větší chunk cache použijeme, tím více bude zabírat paměti a proto bychom neměli volit zbytečně moc velkou velikost chunk cache.



Obrázek 11: Princip chunk cache (obrázek z dokumentace k HDF5)

V souvislosti s chunk cache je třeba zmínit ještě hashovací tabulku, která ke každé chunk cache přísluší. V této hashovací tabulce se ukládají odkazy na aktuálně načtené chunky. I velikost hashovací tabulky je nastavitelná a je důležité ji nastavit správně. Důvod je jednoduchý, mohlo by se totiž stát, že během I/O bychom museli zahodit z chunk cache chunk z důvodu, že má přiřazené stejné pole v hashovací tabulce a to i když bychom měli místo v chunk cache. Naštěstí zvětšit tuto hashovací tabulku nemá téměř žádné nevýhody, jedinou nevýhodou je mírné zvětšení velikosti datasetu. Doporučení pro velikost hashovací tabulky je zvolit ji jako prvočíslo alespoň stokrát větší než počet chunků, které se najednou vejdou do chunk cache.

4.2.2 Komprese

Jak už bylo zmíněno dříve v této kapitole, tak komprese v HDF5 je možná jen pro datasety uložené po chuncích. To je způsobeno tím, že HDF5 není schopno číst z komprimovaného souboru přímo, ale musí ho nejdříve dekomprimovat. To by v případě kontinuálních datasetů bylo velice náročné. V případě ukládání po chuncích se ale situace velmi zjednodušuje, jelikož je možné

chunk dekomprimovat a uložit do chunk cache, následně provést všechny operace nad daným chunkem a znovu chunk zkomprimovat a uložit na disk.

HDF5 obsahuje pro kompresi řadu kompresních metod, ze kterých si uživatel může vybrat. Uživatel má také možnost použít vlastní kompresní metodu, kterou si pro HDF5 vytvoří. Některé takto vytvořené metody jsou volně dostupné k stáhnutí na internetu. Další výhodou HDF5 je, že nám kompresní metody umožňuje libovolně kombinovat.

První a nejuniverzálnější z kompresních metod, které HDF5 obsahuje je metoda gzip (viz [26]). Tato metoda využívá knihovnu zlib, která implementuje známý kompresní algoritmus deflate (viz [8]). Metoda gzip nabízí dobrý kompresní poměr pro širokou škálu dat aniž by potřebovala velké množství systémových zdrojů.

Druhou kompresní metodou, kterou můžeme použít je metoda Szip (viz [12, kapitola 5.6.3]). Tato metoda je založená na Ricově kódování (viz [10]) a funguje tak, že ukládanou hodnotu celočíselně vydělíme nějakým zvoleným parametrem násobnosti 2. Výsledek celočíselného dělení se uloží v unárním kódování, zbytek ve zkráceném binárním kódování. Tyto dvě kódování dobře ilustruje obrázek 12. Szip komprese dosahuje podobného kompresního poměru a času dekomprese jako gzip, ale dosahuje výrazně lepších časů u samotné komprese. Její nevýhodou je, že na rozdíl od gzip není tak univerzální, dá se použít jen na základní atomické datové typy.

Unární kódování	
q	**výsledné bity**
0	0
1	10
2	110
3	1110
4	11110
5	111110
6	1111110
⋮	⋮
N	$\underbrace{111\cdots111}_{N}0$

Zkrácené binární kódování			
r	**ofset**	**binární**	**výsledné bity**
0	0	0000	000
1	1	0001	001
2	2	0010	010
3	3	0011	011
4	4	0100	100
5	5	0101	101
6	12	1100	1100
7	13	1101	1101
8	14	1110	1110
9	15	1111	1111

Obrázek 12: Unární kódování (vlevo) a zkrácené binární kódování (vpravo) u Szip komprese (obrázek z [10])

Další kompresní metodou je metoda N-bit (viz [12, kapitola 5.6.1]), jejíž princip spočívá v tom, že se u hodnot neukládají nevyužité bity. Například 12-bitový integer je v paměti reprezentován nejméně 2 byty, ale pro jeho uložení stačí jen 12-bitů. I tato metoda není univerzální, je možné ji použít jen na datové typy odvozené od typů integer a float.

Poslední kompresní metodou je metoda Scale-offset (viz [12, kapitola 5.6.2]). I tato metoda má omezení co se týče datových typů, funguje totiž jen pro datové typy int, float a double. Pro

int se nejdříve zjistí minimální hodnota v chunku, kterou si HDF5 uloží a následně ukládá jen rozdíl od této hodnoty. Hodnoty float a double nejdříve převede na int a ty následně zkomprimuje stejným způsobem. V tomto případě je komprese ztrátová. Uživatel má taky možnost zadat maximální počet bitů, které budou použity pro uložení rozdílu. V takovém případě se může stát, že uložení bude rovněž ztrátové.

4.3 Přístup k datům

Spousta informací o přístupu k datům byla již zmíněna v předchozích částech v souvislosti s různými možnostmi pro ukládání dat. Proto se v této části budeme v některých chvílích na tyto informace odkazovat.

Pro datasety kontinuálního typu a typu po chuncích umožňuje HDF5 přímý přístup k datům. Knihovna nám nabízí dvě metody jak k datům přistupovat. První metodou se přistupuje k jednotlivým bodům v datasetu pomocí jejich souřadnic. K těmto bodům nemusíme přistupovat jednotlivě, je možné během jedné operace přistupovat k více bodům zároveň.

Druhá metoda slouží k hromadnému přístupu ke všem bodům určitého hyperobdelníku. Těmto hyperobdelníkům se v HDF5 říká hyperslaby. Jednotlivé hyperslaby lze i kombinovat a můžeme tak přistupovat k různým komplexním útvarům bodů. Pomocí hyperslabů lze vytvořit i různé přístupové vzory a přistupovat tak například ke každému třetímu bodu najednou.

Na samotný výkon operací má vliv spousta faktorů jako jsou velikost dat a zvolený typ ukládání. Pokud použijeme ukládání po chuncích tak je také důležité nastavit správnou velikost chunků, chunk cache i hashovací tabulky. Vliv na výkon má samozřejmě i případná komprese a její typ. O všech těchto vlivech bylo již psáno v předchozích částech.

Další možností, kterou nám HDF5 umožňuje zvolit pro optimalizaci přístupu, jsou ovladače pro přístup k datům. Tyto ovladače mapují adresový prostoru HDF5 a úložiště. HDF5 obsahuje řadu těchto ovladačů a také umožňuje uživateli použít vlastní ovladače. Standardně HDF5 používá ovladač H5FD_SEC2, který mapuje adresový prostor přímo do jednoho souboru. Dalším ovladačem je ovladač H5FD_MPIIO, který se využívá v paralelní verzi knihovny.

Pro využití pro paralelní aplikace má HDF5 formát i jisté nevýhody. HDF5 totiž nepodporuje souběžný zápis více uživatelů do souboru. V nynější nejnovější verzi neumožňuje ani souběžné čtení a zápis. Jediné co HDF5 momentálně umožňuje je souběžné čtení více uživatelů. Od nové verze by ale HDF mělo podporovat souběžné čtení více uživatelů a zápis jednoho uživatele.[10]

[10] Více na: https://www.hdfgroup.org/HDF5/docNewFeatures/ [citováno 24.dubna 2016]

5 Nadstavba nad HDF5 pro ukládání časových řad

Přesto, že je HDF5 vytvořeno pro ukládání velkých dat a použití na HPC clusterech, pro jeho využití k ukládání časových řad je třeba jistých úprav. HDF5 totiž postrádá indexování a vyhledávání pomocí času, funkce která je pro ukládání časových řad velmi důležitá. HDF5 nám také nabízí jen rozhraní na jeho použití, pro vytvoření samostatné aplikace, která HDF5 bude využívat je tedy třeba jisté úsilí.

V této části se budeme zabývat knihovnou pro ukládání časových řad, která vznikla jako nadstavba nad HDF5. Tato knihovna je součástí většího projektu pro ukládání a přenos dat Národního superpočítačového centra IT4Innovations. Součástí projektu je také práce Petra Zubka, který pro tuto knihovnu vytvořil adaptéry pro její použití v různých technologiích jako jsou C# nebo Excel (viz [15]).

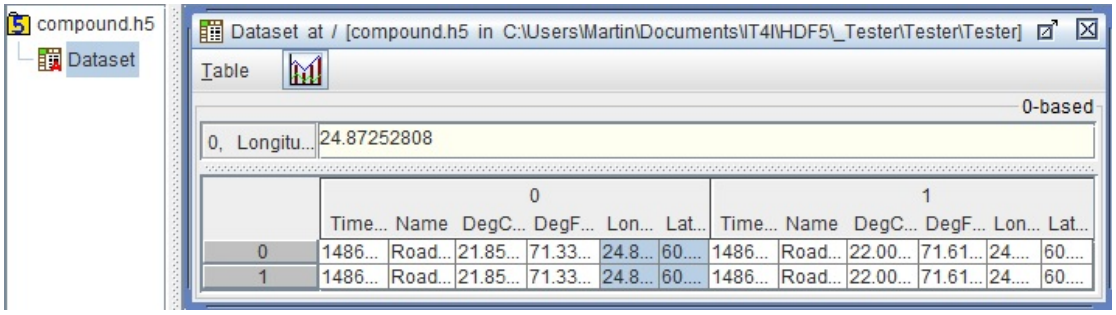
5.1 Popis knihovny

Pro implementaci knihovny byl zvolen programovací jazyk C++, který je široce využíván na HPC clusterech pro svou rychlost a dobrý výkon. Tento jazyk je výhodný i díky své multiplatformnosti a také je to jeden z jazyků podporovaných HDF5.

Data v této knihovně se ukládají do dvourozměrných datasetů. Jednotlivé řádky značí záznamy v čase, jednotlivé sloupce pak jednotlivé časové řady z různých měřících stanic. Vzhledem k tomu, že se ukládají velké časové řady, u kterých se předpokládá, že budou s postupem času narůstat, tak se v této knihovně vždy používá ukládání po chuncích. Můžeme zvolit i vlastní velikosti chunků a optimalizovat tak přístup k datům. Knihovna také umožňuje využívat vlastnosti chunkovaných datasetů jako jsou rozšiřování datasetů a komprese. Zatím jedinou kompresní metodu, kterou knihovna umožňuje používat, je metoda gzip. Tuto metodu je možno používat v devíti různých úrovních. První úroveň je nejrychlejší, ale má nejmenší kompresní poměr. Devátá úroveň má naopak nejlepší kompresní poměr, ale je nejpomalejší.

Knihovna také nabízí dva způsoby pro ukládání samotné struktury dat. První metoda spočívá v tom, že celou strukturu uložíme do jednoho dvojrozměrného datasetu a každé pole tohoto datasetu bude mít datový typ celé struktury. Takto vytvořený dataset budeme dále nazývat složený dataset. Jednoduchý příklad pro složený dataset a strukturu Weather, zobrazený pomocí programu HDFView, můžeme vidět na obrázku 13. Struktura Weather je jedna ze dvou struktur pomocí kterých byla tato knihovna testována. Tato struktura bude popsána v kapitole 6.

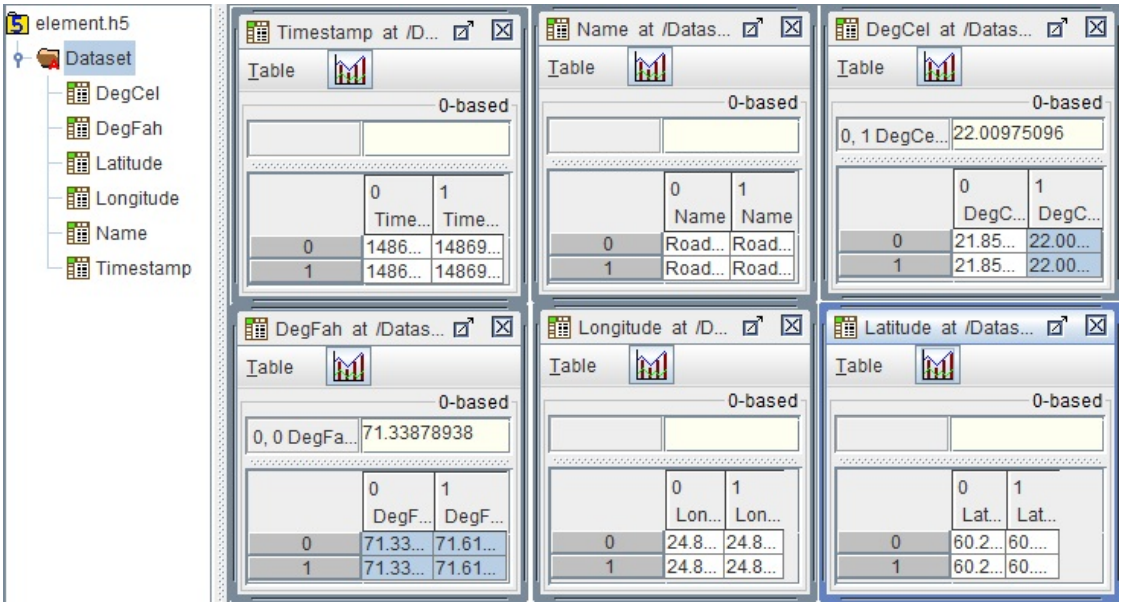
Druhá metoda spočívá v tom, že nejdříve celou strukturu rozdělíme na jednotlivé proměnné a následně každou proměnnou uložíme do vlastního datasetu. Všechny tyto datasety pak seskupíme do jedné groupy pojmenované jako původní dataset. O takto uložených datech budeme dále hovořit jako o rozděleném datasetu. Knihovna pro rozdělené datasety používá stejné metody jako pro složené datasety a není tak třeba využívat jiný interface. Na obrázku 14 můžeme vidět tento typ uložení znovu pro jednoduchý příklad se strukturou Weather.



Obrázek 13: Uložení struktury pomocí jednoho datasetu

Jednou z motivací pro přístup, kdy se ukládá každá proměnná struktury zvlášť, byla možnost přidání další proměnné do struktury až v průběhu času. Pokud totiž ukládáme celou strukturu do jednoho datasetu, tak přidání další proměnné možné není. Neumožňuje to HDF5, ve kterém musí mít celý dataset stejný datový typ a při přidání další proměnné by se tedy musely data ukládat do jiného datasetu. V případě ukládání každé proměnné do zvláštního datasetu tento problém není, jelikož jednoduše vytvoříme nový dataset jen s touto proměnnou. Tuto funkcionalitu tedy HDF5 umožňuje, ale zatím není v knihovně implementovaná. Rozdíl v efektivnosti mezi těmito dvěma metodami prozkoumáme pomocí testů v kapitole 6.

Knihovna díky svým metodám usnadňuje ukládání časových řad v HDF5, ale má jisté omezení. Je navrhnuta pro časové řady s konstantním časovým krokem mezi jednotlivými záznamy. Tyto časové řady nemusí být ucelené - knihovna také umožňuje ukládat řady s chybějícími záznamy. V knihovně můžeme ukládat i časové řady s různými časovými kroky, ale v takovém případě nemůžeme k datům přistupovat na základě času, ale jen na základě jeho souřadnic.



Obrázek 14: Uložení každé proměnné struktury do vlastního datasetu

Stejně tak je možné v knihovně uložit data, která nemají časovou složku. Ale v tomto případě znovu platí, že k datům můžeme přistupovat jen pomocí souřadnic.

5.2 Fungování knihovny

Pro vytvoření datasetů a atributů je třeba zadat do HDF5 jejich datový typ. K vytvoření tohoto datového typu slouží v knihovně struktura StructInfo, která má několik virtuálních funkcí, pomocí kterých knihovna zjistí jaký datový typ vytvořit. Tuto strukturu můžeme vidět ve výpisu 1.

```
struct StructInfo
{
public:
   //Vraci pocet parametru struktury
   virtual int getNumOfParameters() = 0;

   //Vraci velikost struktury
   virtual int getSize() = 0;

   //Vraci string se jmeny parametru struktury oddelenymi strednikem
   virtual std::string getNames() = 0;

   //Vraci string s datovymi typy parametru struktury oddelenymi strednikem
   virtual std::string getTypes() = 0;

   //Vraci string s realnymi bitovymi offsety jednotlivych promennych ve
      strukture
   virtual std::string getOffsets() = 0;

   //Vraci nazev struktury
   virtual std::string getStructureName() = 0;
};
```

Výpis 1: struktura StructInfo

První funkcí struktury StructInfo je funkce *getNumOfParameters*, která slouží k získání počtu proměnných ukládané struktury. Pomocí funkce *getSize* knihovna získává informaci o velikosti struktury v paměti. Funkce *getNames* slouží k získání názvů jednotlivých proměnných. Návratová hodnota této funkce je typu string a jednotlivé názvy proměnných jsou odděleny středníkem.

Další funkcí, kterou struktura má, je funkce *getTypes*, pomocí které se knihovně předávají jednotlivé datové typy proměnných. Tyto datové typy jsou znovu předány pomocí stringu odděleného středníkem. Knihovna momentálně umožňuje ukládat proměnné těchto jednoduchých datových typů: char, signed char, unsigned char, int, short, long, long long, unsigned int, unsigned short, unsigned long, unsigned long long, float a double. Zároveň také můžeme ukládat pole konstantní nebo proměnné délky těchto jednoduchých datových typů. Pro jednoduché datové typy se do metody *getTypes* zapíše jednoduše jejich typ, pro pole konstantní délky se zapíše typ a délka pole oddělena čárkou. Nakonec pro pole proměnné délky se zapíše typ a označení VL oddělené čárkou. Pokud bychom tedy například chtěli uložit strukturu o třech proměnných int, pole floatů délky 5 a pole shortů proměnné délky, tak by návratová hodnota funkce *getTypes* vypadala takto: "int;float,5;short,VL".

Funkce *getOffsets* slouží k získání bytových ofsetů jednotlivých proměnných struktury. Stejně jako u předchozích metod *getNames* a getTypes používá i tato metoda jako návratovou hodnotu string, ve kterém jsou offsety jednotlivých proměnných odděleny středníkem. Pro vypočítání můžeme použít makro HOFFSET, které je součástí HDF5 knihovny.

Poslední funkcí, kterou struktura StructInfo obsahuje je funkce *getStructureName*, která vrací jednoduchý string s názvem této struktury. Tento string se během vytvoření datasetu nebo atributu uloží do knihovnou skrytého atributu a slouží k zpětnému rozpoznání datové struktury. Tento skrytý atribut, který se vždy vytvoří ke každému datasetu a atributu slouží knihovně zároveň i k rozpoznání složených a rozdělených datasetů.

5.3 Funkce knihovny

Knihovna obsahuje dva hlavičkové soubory s veřejnými funkcemi. V hlavičkovém souboru IO.h jsou funkce pro vytvoření, zápis, čtení a mazání datasetů, group a atributů. V souboru Info.h se pak nachází funkce díky kterým můžeme získat další informace o souborech, datasetech groupách i atributech.

5.3.1 I/O funkce

Pro vytvoření groupy můžeme použít funkci *createGroups*, která vytvoří celou strukturu group, pokud tato už neexistuje. Uživatel si také může zvolit jestli metoda tuto strukturu vytvoří v existujícím souboru, nebo vytvoří soubor nový. Fungování této funkce spočívá v tom, že se nejdříve vytvoří nebo otevře soubor v závislosti na volbě uživatele. Následně se od kořenové groupy rekurzivně prochází požadovaná struktura group a pokud aktuálně hledaná groupa neexistuje, tak se tato groupa vytvoří.

Funkce *create* slouží k vytvoření datasetu, nastavení struktury datasetu, velikosti jeho chunků a komprese. V této funkci si také můžeme zvolit jestli chceme data ukládat do složeného nebo rozděleného datasetu. Funkce nejdříve zavolá metodu *createGroups* a dá se tak použít i pro vytvoření souboru a struktury group. Následně se vytvoří buď jeden nebo více rozšiřitelných

datasetů v závislosti na metodě ukládání, kterou chceme použít a nastaví se jim požadovaná velikost chunků a komprese.

Pro zapisování do datasetů knihovna obsahuje dvě funkce. První funkce *append* slouží k zápisu bloku dat na začátek nového řádku v datasetu. Funkce tento blok zapisuje postupně po řádcích, vždy zapíše celý řádek pomocí jednoho volání HDF5 knihovny. Tato funkce byla navrhnuta pro postupné ukládání časových záznamů.

Druhou zapisovací funkcí je funkce *writeAtLoc*, která zapíše blok dat na přesné místo v datasetu. Může se použít pro úpravu starých záznamů, nebo pro přidání dalších sloupců (časových řad) do datasetu. Je schopná vytvořit i mezery v datasetu, pokud některé údaje chybí. Stejně jako funkce *append* i tato funkce zapisuje data po jednotlivých řádcích.

Poslední zapisovací funkcí je funkce *writeAttribute*, která jak už z názvu vyplývá slouží k zapisování atributů. Tato funkce zároveň atribut vytvoří, pokud ještě neexistuje. V případě, že existuje tak do atributu jen zapíše nová data.

Pro čtení nabízí knihovna velké množství funkcí. Funkce *readBlock* slouží pro čtení uceleného bloku celé struktury dat. Tato funkce má dvě varianty, mezi kterými si můžeme zvolit, první varianta čte data po řádcích, druhá po sloupcích. V závislosti na tom jakou variantu si zvolíme, tak se mění i vracené pole. Pokud čteme po řádcích, tak se do pole za sebou zapisují jednotlivé řádky, pro čtení po sloupcích se za sebe zapisují jednotlivé sloupce. Podobně funguje i funkce *readBlockParam*, rozdíl mezi těmito funkcemi je v tom, že *readBlockParam* nečte celou strukturu dat, ale jen jednu proměnnou z nich.

Další metodou pro čtení je funkce *readPoints*, která podle zadaných souřadnic přečte celou strukturu dat na těch souřadnicích, které se jí zadají. Tyto souřadnice můžou být libovolné a během jednoho volání můžeme číst libovolné množství dat. Funkce pak vrátí ucelené pole dat a tyto data budou ve stejném pořadí, v jakém jsou hledané souřadnice. Znovu podobně funguje metoda *readPointsParam*, která je variantou *readPoints* pro čtení jen jedné proměnné struktury.

Poslední funkcí pro čtení je *readAttribute*, která slouží pro čtení dat v atributu. Tato funkce vždy přečte celý atribut a nedá se tedy u ní použít částečný přístup. Toto je omezení způsobené už HDF5 knihovnou, která pro atributy částečný přístup neumožňuje.

Knihovna obsahuje také dvě funkce pro mazání objektů. První funkce se nazývá *deleteAttribute* a slouží k mazání atributů. Pro mazání datasetů a group, pak slouží funkce *deleteObject*. Tato funkce vymaže odkaz na tento objekt, který se tak stane nedostupným, ale nedokáže vymazat data tohoto objektu ze souboru. Důvod tohoto omezení je v tom, že tuto funkcionalitu HDF5 neumožňuje. Pro definitivní smazání, by bylo třeba vytvořit nový HDF5 soubor bez smazaných objektů. K tomuto zkopírování dat je možné použít nástroj příkazové řádky jménem H5repack, který je součástí HDF5.

5.3.2 Informační funkce

Pro získání seznamu datasetů, které jsou uložené v HDF5 souboru slouží funkce *getFileDatasets*. Tato funkce vrací string, ve kterém jsou jednotlivé položky seznamu oddělené středníkem. Každá

položka seznamu obsahuje cestu HDF5 strukturou a informaci o tom jestli se jedná o složený nebo rozdělený dataset, obojí oddělené čárkou.

Druhou funkcí pro získání informací o obsahu souboru je funkce *getGroupItems*, která jak už z názvu vyplývá slouží k získání seznamu objektů, které obsahuje konkrétní groupa. Stejně jako u předchozí funkce se vrací string rozdělený středníky na jednotlivé objekty. U každého objektu se pak vrátí název a typ objektu, znovu oddělené čárkou. V tomto případě může být rovněž typem objektu složený, nebo rozdělený dataset, ale i jiná groupa, nebo atribut.

Poslední funkcí k získání obsahu souboru je funkce *getDatasetAttributes*. Tato funkce slouží k získání seznamu atributů konkrétního datasetu. Funguje podobným způsobem jako předchozí dvě funkce a znovu vrací string oddělený středníky. Mezi středníky pak nalezneme název atributu a typ objektu, v tomto případě může být typem objektu jenom atribut.

K zjištění jestli je dataset složený nebo rozdělený můžeme mimo funkcí *getFileDatasets* a *getGroupItems* použít i funkci *getDatasetType*. Tato funkce je vždy pro jeden dataset a vrací string s informací, jestli je tento dataset složený nebo rozdělený.

Pro získání názvu struktury, která je v datasetu uložená, slouží funkce *getDatasetStructure*, která vrací string s názvem této struktury. Stejně funguje i funkce *getAttributeStructure*, která slouží k získání názvu struktury uložené v atributu.

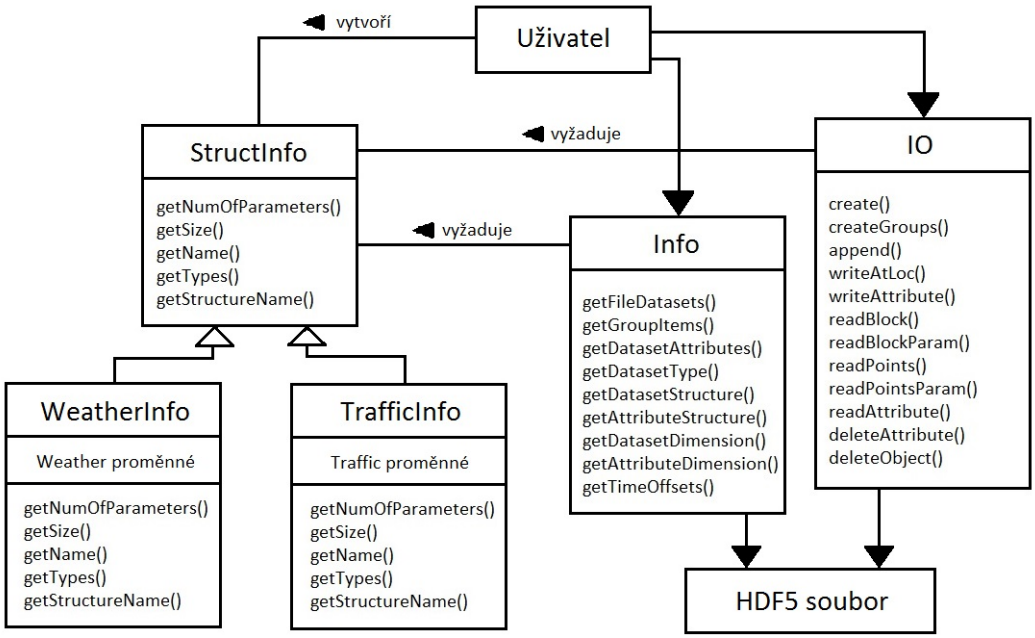
Pokud chceme zjistit velikosti dimenzí datasetu, můžeme použít funkci *getDatasetDimension*. Tato funkce požaduje jako jeden z parametrů odkaz na pole, do kterého jednotlivé velikosti vrátí. První prvek pole bude obsahovat počet řádků, druhý prvek počet sloupců datasetu. Funkce *getAttributeDimension* funguje úplně stejným způsobem, ale pro získání velikosti dimenzí atributů.

Poslední informační funkcí je funkce *getTimeOffsets*, která slouží k získání časového rozestupu mezi jednotlivými záznamy datasetu. Toto pak díky tomu, že se ukládají jen řady s konstantním časovým krokem, slouží k jednoduchému zjištění na kterém řádku se nachází konkrétní časový záznam.

5.4 Použití knihovny

Knihovna je jednoduchá pro používání, model jejího použití můžeme vidět na obrázku 15. Nejdříve je potřeba vytvořit strukturu reprezentující reálná data, která bude dědit ze struktury StructInfo popsané v této kapitole a implementovat její metody. Struktura pak zároveň bude sloužit i pro samotné I/O operace, kterým se předá pole této struktury a ty ho buď uloží v případě zápisu, nebo naplní v případě čtení.

Následně už nám nic nebrání k použití funkcí knihovny k uložení dat. Nejdříve je třeba vytvořit dataset pomocí funkce *create*, případně si můžeme vytvořit strukturu group pomocí funkce *createGroups*. K zápisu datasetu můžeme využít funkci *append* pro zapsání dat na konec datasetu. Pokud potřebujeme zvolit přesnou pozici zápisu tak nám poslouží metoda *writeAtLoc*. K zápisu metadat a jednotlivých proměnných slouží funkce *writeAttribute*, která tyto malé data uloží do atributů. Pro čtení bloku dat můžeme využít funkci *readBlock*, pro čtení jednotlivých bodů funkci *readPoints*. Pokud chceme číst jen jednu proměnnou struktury a ne celou strukturu,



Obrázek 15: Model použití knihovny pro ukládání časových řad v HDF5

tak nám poslouží funkce *readBlockParam* a *readPointsParam*. Pro čtení dat z atributů slouží funkce *readAttribute*. V případě, že chceme z HDF5 souboru smazat atribut, tak nám poslouží funkce *deleteAttribute*. Pro smazání group a datasetů pak slouží funkce *deleteObject*. Pro získání dodatečných informací o datasetech, groupách a atributech můžeme použít informační funkce, které nalezneme v hlavičkovém souboru Info.h.

Detailnější popis použití knihovny, jednotlivých metod a jejich parametrů se nachází v dokumentaci, která byla ke knihovně vytvořena. Tuto dokumentaci nalezneme v příloze B.

6 Experimenty

Knihovna byla otestována na rozsáhlé sadě testů, které budou v této kapitole popsány. Byl testován zápis, čtení a velikosti HDF5 souborů. U testů čtení se testovaly různé přístupy, konkrétně čtení bloků dat, čtení sloupců a řádků a čtení náhodných bodů, to vše pro celou strukturu i jednu proměnnou struktury. Tyto testy byly provedeny pro různé velikosti chunků a různé úrovně komprese. Speciálním testem pak byl test paralelního přístupu k HDF5 souboru z většího počtu procesů, který by měl otestovat jak dobře zvládá HDF5 soubor přístup více uživatelů. Všechny testy proběhly nad testovacími daty Traffic a Weather, které budou popsány v následující části. Vzhledem k rozsáhlosti testů budou v této práci prezentovány vždy jen ty nejzajímavější výsledky a u ostatních bude jen zmíněno proč tomu tak je. Kompletní výsledky pak můžeme nalézt v příloze C.

Pro testování byly použity výpočetní uzly bez akcelerátoru superpočítače Salomon. Každý z těchto uzlů obsahuje dva dvanáctijádrové procesory Intel Xeon E5-2680v3 (2,5GHz) a 128GB RAM (viz [6]). Testy byly spouštěny v adresáři work, který používá souborový systém Lustre a má propustnost 30GB/s (viz [25]). Většina testů probíhala na jednom uzlu ve skupinkách po 24 testech. Každý test tak měl k dispozici jedno jádro výpočetního uzlu. Výjimkou byl test paralelního přístupu více procesů k HDF5 souboru, který proběhl i na více uzlech, ale i v tomto případě měl každý dílčí test k dispozici jenom jedno jádro.

6.1 Testovací data

Pro testování byla použita data vygenerovaná generátorem Petra Zubka (viz [15, strana 67]). Tento generátor generuje data pro struktury Traffic a Weather. Data nejsou generovaná úplně náhodně, snaží se simulovat reálnou situaci. To znamená, že například při generování rychlosti u struktury Traffic se tato rychlost mění plynule a může tak simulovat reálné změny v dopravě.

Struktura Traffic vznikla na základě dopravních dat společnosti HERE (viz [13]). O této struktuře se ve své práci hodně rozepisuje Petr Zubek (viz [15, strana 45-50]) a tak zde budou zmíněny jen jisté základní informace. Ukládaná struktura dosahuje velikosti 208 bytů a obsahuje těchto 18 proměnných: CREATED_TIMESTAMP, EBU_COUNTRY_CODE, TABLE_ID, LI, DE, PBT, PC, LE, SN, CF_CN, CF_FF, CF_JF, CF_SP, CF_SU, CF_TY, PF_UT, PF_SP a SHP_FC.

Proměnná CREATED_TIMESTAMP udává čas, kdy byl záznam vytvořen a v knihovně je ukládán ve formátu Unix Timestamp. Proměnná EBU_COUNTRY_CODE slouží k uložení kódu země a proměnná TABLE_ID udává kód tabulky, do které spadá příslušný EBU_COUNTRY_CODE. Proměnná LI je unikátní identifikátor pro daný silniční úsek. Do proměnná DE se ukládá textový popis úseku a proměnná PBT obsahuje datum měření. Dále proměnná PC obsahuje TMC lokační kód pro zpětnou kompatibilitu, proměnná LE slouží k ukládání délky silničního úseku od předchozího bodu a v proměnné SN se ukládá informace o poslední změně dat. Proměnná CF_CN udává procentuální věrohodnost dat daného úseku a

proměnná CF_FF nese informaci o rychlosti průjezdu v případě, že je cesta úplně volná. Proměnná CF_JF udává úroveň zácpy úseku v intervalu od 0 do 10. Proměnné CF_SP a CF_SU udávají aktuální rychlosti na úseku, v případě CF_SP je aktuální rychlost shora omezena rychlostním limitem. Proměnná TY je získána z lokační reference a ve většině případů dosahuje hodnoty TR. Proměnné PF_UT a PF_SP slouží k předpovědi rychlosti na úseku, proměnná PF_SP udává předpovězenou rychlost a proměnná PF_UT dobu, po kterou tato předpověď platí. Poslední proměnnou je proměnná SHP_FC, ve které se ukládá počet bodů, definujících křivku úseku.

Struktura Weather je jednodušší, obsahuje 6 proměnných a dosahuje velikosti 88 bytů. Tato struktura slouží k ukládání vygenerovaných dat o počasí, konkrétně se zaměřuje na teplotu. Její první proměnnou je proměnná TIMESTAMP, která slouží k ukládání časového razítka. Stejně jako u struktury Traffic se i u této struktury čas ukládá ve formátu Unix Timestamp. Další proměnnou je NAME, která udává název měřící stanice. Do proměnné DEGCEL se ukládá teplota ve stupních Celsia, proměnná DEGFAH obsahuje teplotu ve stupních Fahrenheita. Poslední dva atributy LON a LAT, udávají GPS souřadnice měřící stanice.

6.2 Testování rychlosti zápisu

Prvním testem, který v rámci této práce proběhl byl test rychlosti zápisu dat. Testoval se zápis bloků o velikostech 1000×1000, 2000×2000, 5000×5000, 10000×10000, 20000×20000 a 50000×50000. Pro zápis byla zvolena funkce *append* a zapisovalo se po blocích o velikosti 200 řádků. Každý test měl k dispozici vlastní soubor, do kterého zapisoval. Tento test proběhl jak pro složené, tak i rozdělené datasety. Během testu bylo otestováno pět různých velikostí chunků a to 10×10, 50×50, 100×100, 100×1 (řádek) a 1×100 (sloupec). Nakonec byl otestován i zápis ve spojitosti s kompresí, pro tento test byly zvoleny velikosti chunků 100×100 a byla testována komprese gzip v úrovních 2, 4, 6, 8 a 9.

6.2.1 Výsledky pro různé chunky

Výsledky vybraných zajímavých testů pro strukturu Weather můžeme vidět v tabulce 1. Nejlepších výsledků dosahovalo chunkování o velikosti 100×1, tedy chunkování po řádku. Tento výsledek rozhodně není překvapující, když si uvědomíme, že funkce *append* zapisuje data rovněž po řádcích. Naopak nejhorších výsledků dosahovalo chunkování po sloupcích, tedy 1×100. Pro největší datasety bylo toto chunkování 60krát horší než chunkování po řádcích. Protože tento typ chunkování byl podle očekávání výrazně nejhorší i v ostatních případech, tak ho pro tento test dále nebudu uvádět.

Pokud se nebudeme dívat na speciální případy chunkování po řádcích a sloupcích, tak zjistíme, že pro složené datasety dopadlo nejlépe chunkování 10×10, zatímco pro rozdělené datasety chunkování 50×50. Toto může mít jednoduché vysvětlení a to je to, že rozdělené datasety obsahují pro každou proměnou struktury Weather vlastní dataset. Díky tomu jsou tyto datasety a

rovněž jejich chunky mnohem menší než je tomu u datasetů složených a proto dosahují s chunky větších rozměrů lepších výsledků než datasety složené.

V poslední řadě také můžeme zjistit, že složené datasety dosahují při zápisu lepších výsledků než rozdělené datasety, pro některé případy jsou až 4krát rychlejší.

	Složený dataset			Rozdělený dataset	
Rozměry	10×10	100×1	1×100	50×50	100×1
1000×1000	$0,65$s	$0,57$s	$20,23$s	$2,12$s	$1,68$s
2000×2000	$8,65$s	$1,84$s	$220,48$s	$8,11$s	$5,25$s
5000×5000	$54,81$s	$9,15$s	$655,01$s	$103,93$s	$46,38$s
10000×10000	$319,84$s	$39,74$s	$3020,52$s	$1224,84$s	$170,02$s
20000×20000	$2388,76$s	$276,66$s	$11867,44$s	$4123,90$s	$709,99$s
50000×50000	$8398,20$s	$1127,30$s	$67639,89$s	$23044,31$s	$4401,19$s

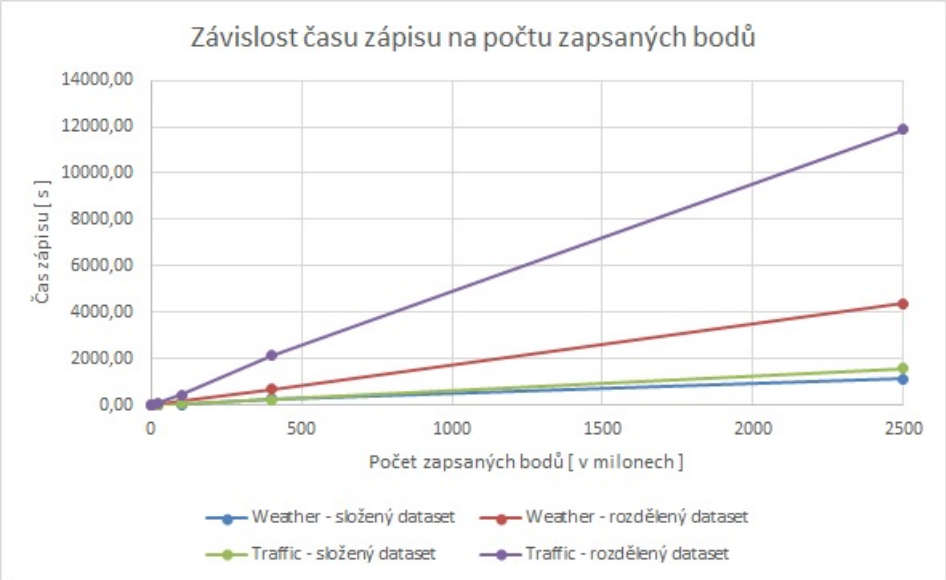
Tabulka 1: Zápis struktury Weather pro různé velikosti chunků

V tabulce 2 můžeme vidět výsledky testu pro strukturu Traffic. Všeobecně nejlepších výsledků opět dosahovalo chunkování 100×1 po řádcích, ale pro menší rozdělené datasety bylo lepší chunkování 100×100. Zajímavé výsledky přineslo testování složených datasetů, kde chunkování 100×100 i chunkování 10×10 bylo mnohem rychlejší než chunkování 50×50, které je mezi nimi. Toto je velmi zajímavý výsledek, který ukazuje, že najít ideální velikost chunků nebude vždy jednoduché. Stejně jako pro strukturu Weather i u této struktury platí, že lepších časů dosahovaly složené datasety. V tomto případě byl rozdíl mezi nimi ještě větší, složené datasety byly až 10krát rychlejší.

	Složený dataset				Rozdělený dataset	
Rozměry	100×100	50×50	10×10	100×1	100×100	100×1
1000×1000	$1,32$s	$8,26$s	$3,90$s	$0,68$s	$3,33$s	$7,49$s
2000×2000	$7,60$s	$82,58$s	$15,09$s	$2,51$s	$12,15$s	$17,69$s
5000×5000	$31,96$s	$345,87$s	$89,57$s	$13,04$s	$119,65$s	$112,67$s
10000×10000	$112,78$s	$2639,83$s	$982,94$s	$76,90$s	$708,42$s	$449,20$s
20000×20000	$534,49$s	$7049,90$s	$4828,48$s	$247,00$s	$3827,15$s	$2127,16$s
50000×50000	$3364,14$s	$28182,83$s	$11374,42$s	$1587,74$s	$26103,96$s	$11863,88$s

Tabulka 2: Zápis struktury Traffic pro různé velikosti chunků

Posledním co by nás mohlo u testu zápisu zajímat je, jak rychle roste čas v závislosti na počtu zapisovaných bodů. Toto sice dokážeme vyčíst i z tabulek, ale pro lepší přehlednost se můžeme podívat na graf 16. Tento graf nám ukazuje tuto závislost pro všechny testy s chunky o velikosti 100×1, tedy s nejlepšími výsledky pro jednotlivé případy. Pro přehlednost jsou rozměry zapisovaných datasetů převedeny na počty bodů v milionech. Jednotlivé body na této ose pak přesně odpovídají rozměrům datasetů, které byly v rámci testů vytvořeny.



Obrázek 16: Graf závislosti času zápisu na počtu zapsaných bodů

Z grafu můžeme vidět, že čas s přibývajícími body narůstá lineárně. Dokonce v některých případech je nárůst času mírně menší než lineární. Tento lineární nárůst se projevil u všech možností, které byly během testu zápisu testovány. Výsledek nám ukazuje, že HDF5 zvládá rychle ukládat i velká data a hodí se tak pro jejich ukládání.

6.2.2 Výsledky pro kompresi

Výsledky vybraných testů komprese pro složené datsety a strukturu Weather můžeme vidět v tabulce 3. Nejlepších časů podle očekávání dosahovala komprese 2. úrovně, nejhorších pak komprese 9. úrovně. Při porovnání s časem zápisu bez komprimování byla komprese 2. úrovně až 9krát pomalejší, zatímco komprese 9. úrovně dosahovala až 20krát horších výsledků. Časy ostatních testovaných úrovní komprese pak byly přibližně rovnoměrně rozprostřené mezi těmito kompresemi.

	Složený dataset		
Rozměry	Bez komprese	2. úroveň	9. úroveň
1000×1000	$4,90$s	$79,46$s	$173,89$s
2000×2000	$51,52$s	$316,85$s	$694,42$s
5000×5000	$367,51$s	$1966,77$s	$4328,63$s
10000×10000	$1109,85$s	$8037,34$s	$17535,10$s
20000×20000	$4024,03$s	$32131,37$s	$69744,14$s
50000×50000	$22256,91$s	$198942,95$s	$434512,57$s

Tabulka 3: Zápis struktury Weather pro různé úrovně komprese (1. část)

V tabulce 4 pak můžeme vidět výsledky komprese pro rozdělené datasety. I u tohoto testu dopadla nejlépe komprese 2. úrovně, která byla přibližně sedmkrát pomalejší než zápis bez komprese. Zajímavé výsledky přinesly testy kompresí 8. a 9. úrovně. Komprese 8. úrovně totiž byla pomalejší než komprese 9. úrovně. Nejpomalejší 8. úroveň dosahovala přibližně 15krát horších výsledků, než zápis bez komprese.

Pokud srovnáme výsledky pro složené a rozdělené datasety, tak zjistíme, že rozdělené datasety dosahují při kompresi lepších výsledků než datasety složené, jsou přibližně o čtvrtinu rychlejší.

		Rozdělený dataset		
Rozměry	Bez komprese	2. úroveň	8. úroveň	9. úroveň
1000×1000	2, 78s	18, 76s	67, 17s	55, 56s
2000×2000	18, 56s	249, 62s	550, 98s	505, 53s
5000×5000	269, 94s	1524, 19s	3408, 13s	3125, 22s
10000×10000	1130, 10s	6245, 62s	13788, 83s	12519, 14s
20000×20000	3069, 64s	24799, 86s	54529, 81s	51692, 64s
50000×50000	22528, 09s	155602, 35s	343422, 39s	312527, 29s

Tabulka 4: Zápis struktury Weather pro různé úrovně komprese (2. část)

Výsledky testu pro strukturu Traffic nalezneme v tabulce 5. I pro tento test dosahovala nejlepších výsledků komprese 2. úrovně. Pro složené datasety byla tato komprese přibližně 80krát pomalejší než zápis bez komprese, pro rozdělené datasety to bylo přibližně osmkrát. U této struktury již nebyly žádné překvapivé výsledky a další úrovně komprese byly vždy o něco pomalejší než komprese předchozí. Také se znovu potvrdilo, že rozdělené datasety dosahují pro kompresi lepších výsledků než datasety složené, byly znovu přibližně o čtvrtinu rychlejší.

	Složený dataset		Rozdělený dataset	
Rozměry	Bez komprese	2. úroveň	Bez komprese	2. úroveň
1000×1000	1, 32s	110, 54s	3, 33s	55, 98s
2000×2000	7, 60s	443, 10s	12, 15s	220, 57s
5000×5000	31, 96s	2741, 72s	119, 65s	1853, 74s
10000×10000	112, 78s	11017, 99s	708, 42s	8129, 08s
20000×20000	534, 49s	43979, 47s	3827, 15s	32288, 04s
50000×50000	3364, 14s	274006, 15s	26103, 96s	203596, 98s

Tabulka 5: Zápis struktury Traffic pro různé úrovně komprese

6.3 Testování velikosti souboru

S testem zápisu popsaném v předchozí části úzce souvisí i test velikostí jednotlivých souborů. Tento test je pak opravdu důležitý pro správné posouzení komprese. Během testu byly použity data vytvořené při testu zápisu a budou zde tedy zobrazeny výsledky pro soubory se složenými

a rozdělenými datasety o rozměrech 1000×1000, 2000×2000, 5000×5000, 10000×10000, 20000×20000 a 50000×50000. V tomto testu se bude pracovat jenom s datasety, které mají velikosti chunků 100×100 a to jak bez komprese, tak i s kompresí. Velikosti souborů pro datasety s jiným chunkováním zde uvedeny nebudou, jelikož rozdíly v jejich velikostech jsou minimální.

	Složený dataset			
Rozměry	Bez komprese	2. úroveň	4. úroveň	9. úroveň
1000×1000	84,0 MB	26,2 MB	24,0 MB	26,6 MB
2000×2000	0,3 GB	0,1 GB	0,1 GB	0,1 GB
5000×5000	2,0 GB	0,6 GB	0,6 GB	0,6 GB
10000×10000	8,2 GB	2,5 GB	2,3 GB	2,3 GB
20000×20000	32,8 GB	9,8 GB	9,0 GB	9,0 GB
50000×50000	205,0 GB	58,1 GB	53,1 GB	53,1 GB

Tabulka 6: Test velikostí struktury Weather pro různé úrovně komprese (1. část)

Výsledky testu pro strukturu Weather můžeme vidět v tabulce 6, která obsahuje výsledky testů pro složené datasety a v tabulce 7, ve které jsou výsledky pro rozdělené datasety. Při srovnání velikostí složených a rozdělených datasetů zjistíme, že rozdělené datasety jsou nepatrně menší než datasety složené. Toto je logicky způsobeno tím, že struktury obsahují i bity, které žádná data neobsahují. Jejich rozdělením na jednotlivé proměnné můžeme tyto nevyužité bity ignorovat a ukládat jen samotná data.

Pokud se podíváme na kompresi, tak zjistíme, že kompresní poměr u struktury Weather je pro složené datasety přibližně $3,5:1$, pro rozdělené datasety je to přibližně $4,5:1$. Toto nadále zvyšuje rozdíl ve velikostech mezi složenými a rozdělenými datasety. Nejmenšího kompresního poměru dosáhla podle očekávání komprese 2. úrovně, ale i přesto je zlepšení při použití ostatních úrovní komprese minimální. Toto vyvolává otázku, jestli se vzhledem k nárůstu času potřebného k zápisu vyplatí používat vyšší úrovně komprese.

	Rozdělený dataset			
Rozměry	Bez komprese	2. úroveň	4. úroveň	9. úroveň
1000×1000	76,3 MB	15,4 MB	14,8 MB	14,4 MB
2000×2000	306,0 MB	77,3 MB	73,8 MB	73,1 MB
5000×5000	1,9 GB	0,4 GB	0,4 GB	0,4 GB
10000×10000	7,5 GB	1,7 GB	1,7 GB	1,6 GB
20000×20000	29,8 GB	6,7 GB	6,5 GB	6,3 GB
50000×50000	187 GB	39,6 GB	38,1 GB	37,3 GB

Tabulka 7: Test velikostí struktury Weather pro různé úrovně komprese (2. část)

Pro strukturu Traffic nalezneme výsledky v tabulce 8. I u této struktury jsou rozdělené datasety menší než datasety složené a to jak bez komprese, tak i s kompresí. Pro složené datasety je kompresní poměr přibližně $12,5:1$, pro rozdělené dokonce $33:1$. Z tohoto výsledku je patrné,

že struktura Traffic je mnohem lépe komprimovatelná než struktura Weather a zvláště pro rozdělené datasety dosahuje komprese dobrých výsledků. I u této struktury se znovu projevilo, že vyšší úrovně komprese mají jen minimální vliv na velikost souborů. Z tohoto důvodu nejsou v této tabulce ani uváděné.

	Složený dataset		Rozdělený dataset	
Rozměry	Bez komprese	2. úroveň	Bez komprese	2. úroveň
1000×1000	199,0 MB	16,7 MB	181,0 MB	5,6 MB
2000×2000	794,0 MB	70,8 MB	722,0 MB	22,1 MB
5000×5000	4,8 GB	0,4 GB	4,4 GB	0,1 GB
10000×10000	19,4 GB	1,6 GB	17,6 GB	0,5 GB
20000×20000	77,5 GB	6,4 GB	70,4 GB	2,1 GB
50000×50000	485 GB	38,5 GB	441 GB	13,4 GB

Tabulka 8: Test velikostí struktury Traffic pro různé úrovně komprese

6.4 Testování rychlosti čtení bloku dat

Prvním testem čtení bylo čtení bloků dat. Během tohoto testu byly ke čtení použity datasety o velikostech 50000×50000 a četly se z nich bloky o rozměrech 1000×1000, 2000×2000, 5000×5000, 10000×10000, 20000×20000 a 50000×50000. Test proběhl znovu pro různé velikosti chunků a otestovaly se i komprimované datasety. Tento test měl dvě fáze, v první fázi testu byla použita funkce *readBlock* a proběhlo čtení celé struktury dat. V druhé fázi byla použita funkce *readBlockParam* a proběhlo čtení jedné proměnné struktury. Ve struktuře Weather byla pro toto čtení zvolena proměnná DEGCEL, u struktury Traffic to byla proměnná PC. Funkce *readBlock* a *readBlockParam* byly otestovány v obou jejich variantách, tedy jak pro čtení po řádcích, tak pro čtení po sloupcích.

6.4.1 Výsledky pro různé chunky

Výsledky testu čtení celé struktury pro strukturu Weather nalezneme v tabulce 9. Při čtení po řádcích dopadlo stejně jako u testu zápisu nejlépe chunkování 100×1, jako druhé nejlepší se projevilo chunkování 50×50. I u tohoto testu se ukázalo jako rychlejší použití složených datasetů, které bylo v některých případech až 2krát rychlejší než použití rozdělených datasetů. Čtení po sloupcích dopadlo nejlépe pro chunky velikosti 100×100, ale i u tohoto nejlepšího případu je čtení po sloupcích mnohem pomalejší než čtení po řádcích.

Pro strukturu Traffic nalezneme výsledky v tabulce 10. I zde dopadlo nejlépe chunkování 100×1, následované chunkováním 50×50. Velmi překvapivé výsledky pak přinesly testy s chunkováním o rozměrech 100×100. Toto chunkování bylo stejně jako u struktury Weather nejlepší pro čtení po sloupcích, ale zde byl tento typ čtení dokonce výrazně lepší než čtení po

	Složený dataset			Rozdělený dataset	
Rozměry	50×50 Po řádcích	100×1 Po řádcích	100×100 Po sloupcích	50×50 Po řádcích	100×1 Po řádcích
1000×1000	0,70s	2,77s	2,18s	1,86s	4,76s
2000×2000	2,72s	4,66s	8,18s	4,34s	5,55s
5000×5000	20,39s	9,12s	49,81s	31,21s	51,51s
10000×10000	134,97s	230,88s	420,90s	237,01s	244,44s
20000×20000	506,81s	390,31s	1662,37s	860,63s	648,09s
50000×50000	3349,61s	939,96s	10032,77s	4780,62s	2066,95s

Tabulka 9: Čtení bloku struktury Weather pro různé velikosti chunků

řádcích. Rozdíl byl způsoben hlavně velmi špatnými časy čtení po řádcích. Tento úkaz se projevil jak u datasetů složených, tak i u datasetů rozdělených.

	Složený dataset				Rozdělený dataset	
Rozměry	50×50 Po řád.	100×1 Po řád.	100×100 Po řád.	100×100 Po sloup.	50×50 Po řád.	100×1 Po řád.
1000×1000	1,61s	3,42s	4,22s	1,13s	4,71s	7,09s
2000×2000	8,74s	8,47s	17,73s	4,03s	19,49s	16,22s
5000×5000	48,61s	67,05s	99,0s	24,09s	83,53s	120,11s
10000×10000	287,96s	538,89s	6918,62s	229,25s	630,86s	644,98s
20000×20000	1144,43s	790,90s	35820,82s	1122,11s	2371,78s	1755,43s
50000×50000	6529,62s	2038,91s	195548,12s	7399,19s	14577,11s	7001,84s

Tabulka 10: Čtení bloku struktury Traffic pro různé velikosti chunků

Výsledky čtení proměnné DEGCEL struktury Weather můžeme vidět v tabulce 11. I během tohoto testu dopadly nejlépe datsety s rozměry chunků 100×1, pro složené datasety byly druhé nejlepší datsety s chunky 50×50, pro rozdělené s chunky 100×100. Rovněž se znovu ukázalo, že při čtení bloku dat je lepší variantou využití čtení po sloupcích. Ale tím nejdůležitějším zjištěním při tomto testu bylo, že rozdělené datasety jsou při čtení jedné proměnné výrazně rychlejší než složené datasety. Toto má velmi jednoduché vysvětlení, během čtení jedné proměnné z rozděleného datasetu se vnitřně čte z jediného datasetu, který obsahuje jenom tuto proměnnou. To je pak logicky rychlejší, než čtení z datasetu, ve kterém je kompletní struktura a proměnné se mezi sebou střídají.

Výsledky čtení proměnné PC struktury Traffic můžeme nalézt v tabulce 12. Znovu dopadlo nejlépe chunkování 100×1, tentokrát následované u složených datasetů chunkováním 10×10. U rozdělených bylo druhé nejlepší chunkování 100×100. Úkaz objevený při čtení bloku celé struktury Traffic se částečně objevuje i zde, ale jen u složených datasetů. Můžeme si tak všimnout, že u složených datasetů a chunkování 100×100 je opět varianta čtení po řádcích velmi pomalá, zatímco varianta čtení po sloupcích je rychlejší než u ostatních chunků. Pro rozdělené datasety

	Složený dataset		Rozdělený dataset	
	50×50	100×1	100×100	100×1
Rozměry	Po řádcích	Po řádcích	Po řádcích	Po řádcích
1000×1000	$0,92$s	$1,77$s	$0,07$s	$0,36$s
2000×2000	$2,70$s	$3,92$s	$0,54$s	$0,70$s
5000×5000	$18,06$s	$8,52$s	$3,23$s	$1,83$s
10000×10000	$150,89$s	$171,57$s	$23,74$s	$27,16$s
20000×20000	$519,48$s	$380,35$s	$92,70$s	$58,30$s
50000×50000	$3170,41$s	$926,78$s	$553,02$s	$205,55$s

Tabulka 11: Čtení bloku jedné proměnné struktury Weather pro různé velikosti chunků

už tomu tak není, varianta po řádcích je pro ně rychlejší. Nutno však podotknout, že i tak toto chunkování nabízí nejrychlejší čtení po sloupcích ze všech ostatních.

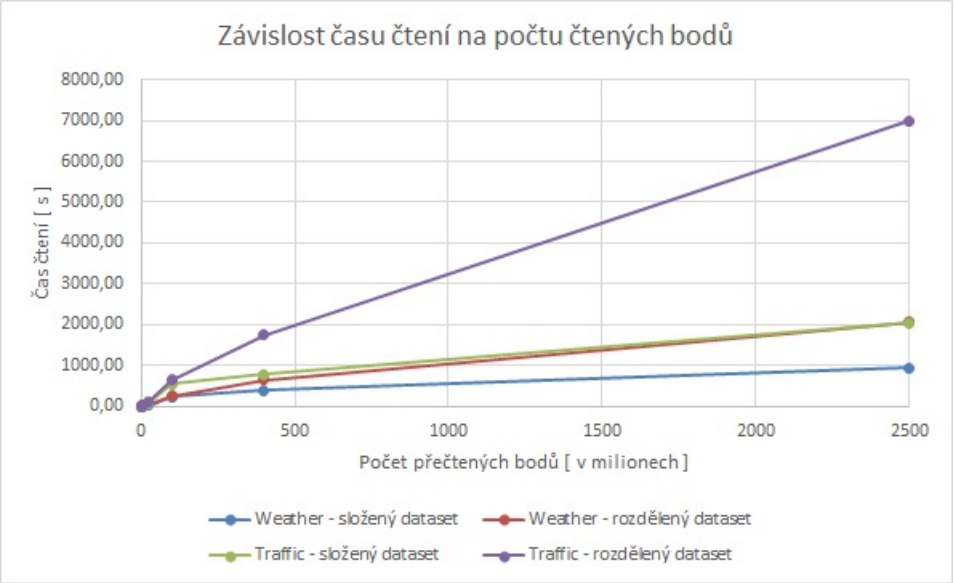
		Složený dataset			Rozdělený dataset	
	10×10	100×1	100×100	100×100	100×100	100×1
Rozměry	Po řád.	Po řád.	Po řád.	Po sloup.	Po řád.	Po řád.
1000×1000	$1,46$s	$3,28$s	$4,67$s	$1,36$s	$0,07$s	$0,24$s
2000×2000	$4,17$s	$6,44$s	$11,00$s	$5,33$s	$0,23$s	$0,51$s
5000×5000	$23,26$s	$60,69$s	$82,11$s	$29,83$s	$2,05$s	$1,39$s
10000×10000	$273,49$s	$387,37$s	$6689,15$s	$195,28$s	$12,63$s	$15,83$s
20000×20000	$939,29$s	$778,52$s	$34894,09$s	$1049,43$s	$45,29$s	$40,23$s
50000×50000	$5115,54$s	$1701,96$s	$203591,28$s	$7171,33$s	$222,93$s	$137,50$s

Tabulka 12: Čtení bloku jedné proměnné struktury Traffic pro různé velikosti chunků

I u testu čtení by nás mohlo zajímat jak roste čas v závislosti na přibývajícím počtu bodů. Tuto závislost nalezneme vyobrazenou na v grafu 17. Stejně jako u zápisu i zde bylo zvoleno chunkování 100×1, které bylo nejrychlejší ve všech testech. Rozměry čtených bloků jsou opět převedeny na počet přečtených bodů. Jak můžeme v grafu vidět i nárůst čtení v závislosti na počtu bodů má převážně lineární charakter. Dokonce se zdá, že čím víc bodů čteme, tím je tento nárůst o trochu nižší.

6.4.2 Výsledky pro kompresi

Výsledky čtení bloku struktury Weather pro kompresi můžeme nalézt v tabulce 13. Stejně jako u zápisu, dosahovaly o něco lepších výsledků datasety rozdělené. Zajímavé výsledky se pak projevily u složených datasetů, kde je komprese 2. úrovně pomalejší, než komprese vyšších úrovní. Všechny ostatní komprese dosáhly přibližně stejných výsledků. U rozdělených datasetů se tento rozdíl mezi kompresí 2. úrovně a ostatními kompresemi neprojevil, všechny dosahovaly přibližně stejných časů.



Obrázek 17: Graf závislosti času čtení bloku dat na počtu přečtených bodů

	Složený dataset				Rozdělený dataset	
	Bez kom.	2.úroveň	2.úroveň	4.úroveň	Bez kom.	2.úroveň
Rozměry	Po řád.	Po řád.	Po sloup.	Po řád.	Po řád.	Po řád.
1000×1000	1,77s	25,54s	25,76s	23,36s	2,07s	7,43s
2000×2000	8,96s	101,94s	101,94s	95,85s	9,30s	83,11s
5000×5000	46,93s	633,24s	633,24s	594,28s	40,09s	510,79s
10000×10000	231,28s	2761,02s	2761,02s	2574,29s	299,30s	2411,95s
20000×20000	779,19s	10639,80s	10639,80s	9899,75s	1030,37s	9029,68s
50000×50000	4379,29s	64892,90s	64892,90s	60777,88s	5436,87s	53410,45s

Tabulka 13: Čtení bloku struktury Weather pro různé úrovně komprese

Pokud srovnáme časy u čtení po řádcích s časy u čtení po sloupcích, tak zjistíme, že pro komprimované datasety dosahují tyto dvě varianty přibližně stejných výsledků. Toto je rozdíl oproti nekomprimovaným datasetům, kde je čtení po sloupcích výrazně pomalejší. Při srovnání časů čtení z komprimovaných datasetů s časy čtení z nekomprimovaných datasetů zjistíme, že pro složené datasety je toto čtení přibližně 16krát pomalejší. U rozdělených datasetů je čtení z komprimovaných datasetů pomalejší 10krát.

Výsledky čtení struktury Traffic z komprimovaných datasetů jsou zobrazeny v tabulce 14. Vzhledem k enormním časům chunkování 100×100 při čtení po řádcích z nekomprimovaného datasetu, je zde pro srovnání uvedena varianta čtení po sloupcích. Tento úkaz se už, ale u komprese neobjevuje, pro složené datasety jsou varianty čtení po řádcích a po sloupcích přibližně rovnocenné. U rozdělených datasetů rozdíl nalezneme, čtení po řádcích je rychlejší než čtení po sloupcích. Pokud se zaměříme na jednotlivé úrovně komprese, tak zjistíme, že vyšší úrovně komprese mají malý, ale pozitivní vliv na rychlost čtení z datasetu. Tento vliv se projevil jak u

složených, tak i rozdělených datasetů.

	Složený dataset		Rozdělený dataset			
	Bez kom.	2.úroveň	Bez kom.	2.úroveň	2.úroveň	8.úroveň
Rozměry	Po sloup.	Po řád.	Po sloup.	Po řád.	Po sloup.	Po řád.
1000×1000	1, 13s	34, 49s	4, 39s	20, 15s	20, 74s	19, 33s
2000×2000	4, 03s	137, 87s	15, 11s	78, 43s	79, 54s	73, 54s
5000×5000	24, 09s	834, 97s	101, 83s	645, 71s	672, 16s	624, 18s
10000×10000	229, 25s	3542, 49s	2280, 06s	3270, 72s	3780, 34s	2974, 44s
20000×20000	1122, 11s	14003, 57s	9479, 63s	12235, 80s	14741, 80s	11428, 32s
50000×50000	7399, 19s	85584, 52s	66998, 09s	72224, 75s	95254, 70s	66503, 71s

Tabulka 14: Čtení bloku struktury Traffic pro různé úrovně komprese

Časy čtení bloku jedné proměnné z komprimovaných datasetů nalezneme v tabulce 15. Během tohoto testu se znovu projevily úkazy známé z testů předchozích. Opět je čtení složených datasetů s kompresí 2. úrovně pomalejší než čtení z datasetů s vyšší úrovní komprese, které dosahují stejných přibližně stejných časů. U rozdělených datasetů jsou časy všech kompresí opět podobné. Velmi dobrých časů pak dosahuje čtení z rozdělených komprimovaných datasetů, u kterých se kombinuje jejich lepší reakce na kompresi a rychlejší čtení jedné proměnné.

	Složený dataset			Rozdělený dataset	
	Bez komprese	2.úroveň	4.úroveň	Bez komprese	2.úroveň
Rozměry	Po řádcích	Po řádcích	Po sloupcích	Po řádcích	Po řádcích
1000×1000	1, 17s	25, 86s	24, 66s	0, 07s	0, 13s
2000×2000	6, 41s	99, 92s	98, 19s	0, 54s	23, 26s
5000×5000	37, 27s	605, 65s	600, 45s	3, 23s	140, 70s
10000×10000	178, 91s	2634, 21s	2582, 13s	23, 74s	648, 28s
20000×20000	687, 77s	10427, 74s	10255, 47s	92, 70s	2514, 82s
50000×50000	4262, 20s	63491, 96s	61510, 27s	553, 02s	15246, 45s

Tabulka 15: Čtení bloku jedné proměnné struktury Weather pro různé úrovně komprese

Výsledky čtení proměnné PC struktury Traffic z komprimovaných datasetů najdeme v tabulce 16. Tyto výsledky znovu potvrdily závěry odvozené pomocí testů minulých. I u tohoto testu byly mnohem rychlejší datasety rozdělené, které lépe reagují jak na kompresi, tak na čtení jedné proměnné. Také si znovu můžeme všimnout, že vyšší úroveň komprese mírně snižuje celkový čas potřebný k čtení datasetu. Toto se opět projevilo u složených i rozdělených datasetů.

6.5 Testování rychlosti čtení řádků a sloupců dat

Druhým testem čtení byl test čtení řádků a sloupců dat. I u tohoto testu se provádělo čtení z datasetů o rozměrech 50000×50000 a byla použita funkce *readBlock* pro čtení celé struktury a *readBlockParam* pro čtení jedné proměnné struktury. I v tomto případě byly zvoleny proměnné

	Složený dataset		Rozdělený dataset		
	Bez komprese	2.úroveň	Bez komprese	2.úroveň	8.úroveň
Rozměry	Po sloupcích	Po řádcích	Po řádcích	Po řádcích	Po řádcích
1000×1000	1, 36s	34, 16s	0, 07s	0, 11s	0, 09s
2000×2000	5, 33s	135, 03s	0, 23s	0, 34s	0, 28s
5000×5000	29, 83s	829, 73s	2, 05s	15, 40s	14, 90s
10000×10000	195, 28s	3508, 56s	12, 63s	75, 94s	82, 04s
20000×20000	1049, 43s	13750, 09s	45, 29s	280, 25s	288, 60s
50000×50000	7171, 33s	84530, 72s	222, 93s	1792, 83s	1640, 03s

Tabulka 16: Čtení bloku jedné proměnné struktury Traffic pro různé úrovně komprese

DEGCEL a PC. Pro čtení řádků byla použita varianta čtení po řádcích, pro čtení sloupců varianta po sloupcích. Vzhledem k tomu, že časy u tohoto testu byly výrazně menší než u čtení bloků dat, bylo testováno jen čtení větších řádků/sloupců o velikostech 10000, 20000 a 50000 bodů. Z tohoto důvodu jsou taky v tabulkách hodnoty v milisekundách a ne v sekundách jak tomu je u ostatních testů.

6.5.1 Výsledky pro různé chunky

V tabulce 17 nalezneme výsledky testu pro komprimované datasety a strukturu Weather. Podle očekávání nejlepších výsledků pro čtení řádků dosahovalo chunkování 100×1, zatímco pro čtení sloupců bylo nejrychlejší chunkování 1×100. Pro opačné operace byla tato dvě chunkování naopak nejhorší a to dost výrazně. Ostatní chunkování dosahovaly vyrovnanějších výsledků, nejlepší z nich pro čtení sloupce z rozdělených datasetů bylo chunkování 100×100, pro všechny ostatní případy chunkování 50×50. Při srovnání časů mezi čtením řádků a čtením sloupců také zjistíme, že čtení řádků je ve většině případů rychlejší než čtení sloupců.

	Čtení řádku					
	Složený dataset			Rozdělený dataset		
Počet bodů	50×50	100×1	1×100	50×50	100×1	1×100
10000	15, 36ms	1, 16ms	97, 87ms	25, 10ms	5, 78ms	513, 16ms
20000	27, 11ms	1, 77ms	201, 66ms	34, 24ms	12, 02ms	1287, 15ms
50000	74, 82ms	4, 66ms	608, 87ms	88, 84ms	28, 32ms	2704, 64ms
	Čtení sloupce					
	Složený dataset			Rozdělený dataset		
Počet bodů	50×50	100×1	1×100	100×100	100×1	1×100
10000	21, 25ms	341, 81ms	6, 88ms	52, 22ms	1683, 01ms	42, 81ms
20000	54, 20ms	731, 04ms	13, 67ms	71, 37ms	3427, 86ms	77, 80ms
50000	118, 90ms	1855, 91ms	33, 22ms	181, 25ms	11053, 44ms	206, 45ms

Tabulka 17: Čtení řádků a sloupců struktury Weather pro různé velikosti chunků

Časy testů pro strukturu Traffic můžeme nalézt v tabulce 18. I u této struktury je chunkování 100×1 nejrychlejší pro čtení řádků a výrazně nejpomalejší pro čtení sloupců. Pro chunkování 1×100 je tomu znovu naopak. Pokud pomineme tyto dvě chunkování, tak ve všech případech dosahovalo nejrychlejších časů chunkování 100×100. Test pro čtení sloupce o velikosti 50000 bodů z rozděleného datasetu o velikostech chunků 1×100 neproběhl, protože při zápisu tohoto datasetu dosáhl test horní hranice doby, po kterou je možné nechat spuštěný test na superpočítači Salomon.

Čtení řádku				
	Složený dataset		Rozdělený dataset	
Počet bodů	100×100	100×1	100×100	100×1
10000	1,96ms	1,32ms	31,21ms	22,95ms
20000	3,49ms	2,36ms	72,58ms	37,79ms
50000	10,64ms	6,53ms	196,39ms	108,64ms
Čtení sloupce				
	Složený dataset		Rozdělený dataset	
Počet bodů	100×100	1×100	100×100	1×100
10000	22,93ms	6,69ms	115,96ms	85,09ms
20000	38,92ms	11,51ms	224,86ms	173,72ms
50000	105,11ms	27,42ms	551,14ms	-

Tabulka 18: Čtení řádků a sloupců struktury Traffic pro různé velikosti chunků

Vzhledem k všeobecné rozsáhlosti testů zde nebudou uvedeny výsledky čtení řádků a sloupců jedné proměnné. I v těchto výsledcích se projevily úkazy, které jsme mohli pozorovat u testů předchozích. Nejlepší a zároveň i nejhorší výsledky byly pro chunky o velikostech 1×100 a 100×1, druhé nejlepší ve všech ohledech pro velikost chunků 100×100. Stejně tak se i u tohoto testu projevilo, že výrazně rychlejších časů u čtení proměnné dosáhneme s rozdělenými datasety. Kompletní výsledky testu můžeme nalézt v příloze C.

6.5.2 Výsledky pro kompresi

Výsledky testu pro kompresi a strukturu Weather nalezneme v tabulce 19. I u tohoto testu čtení dosahují mírně nejhorších výsledků datasety s kompresí 2. úrovně, rozdíly mezi jednotlivými kompresemi jsou ovšem zanedbatelné. Stejně jako u předchozích testů komprese vychází lépe datasety rozdělené, čtení z nich je přibližně o pětinu rychlejší než čtení z datasetů složených. Srovnáním čtení z komprimovaných a nekomprimovaných datasetů také zjistíme, že čtení z komprimovaných je přibližně 10krát pomalejší.

Pro strukturu Traffic jsou výsledky obsaženy v tabulce 20. Na základě dat můžeme vyvodit podobné závěry jako u struktury Weather, tedy rychlejší časy pro rozdělené datasety a velmi malé rozdíly mezi jednotlivými kompresemi. Rozdíl tak najdeme jen v poměru času čtení z

Čtení řádku						
	Složený dataset			Rozdělený dataset		
Počet bodů	Bez kom.	2. úroveň	9. úroveň	Bez kom.	2. úroveň	9. úroveň
10000	28, 89ms	264, 90ms	244, 91ms	36, 22ms	213, 79ms	201, 03ms
20000	52, 73ms	518, 98ms	486, 72ms	49, 21ms	409, 09ms	394, 01ms
50000	114, 98ms	1269, 77ms	1205, 12ms	125, 42ms	1016, 89ms	988, 64ms
Čtení sloupce						
	Složený dataset			Rozdělený dataset		
Počet bodů	Bez kom.	2. úroveň	9. úroveň	Bez kom.	2. úroveň	9. úroveň
10000	32, 72ms	259, 50ms	241, 04ms	52, 22ms	213, 89ms	209, 57ms
20000	50, 32ms	513, 70ms	474, 86ms	71, 37ms	427, 91ms	412, 05ms
50000	128, 70ms	1269, 61ms	1175, 59ms	181, 25ms	1059, 45ms	1012, 71ms

Tabulka 19: Čtení řádků a sloupců struktury Weather pro různé úrovně komprese

komprimovaných a nekomprimovaných datasetů. U struktury Traffic jsou složené komprimované datasety při čtení sloupců přibližně 17krát pomalejší než nekomprimované, při čtení řádků je to dokonce 170krát. Toto ale není způsobeno časy u kompresí, rozdíl je hlavně mezi čtením řádků a sloupců nekomprimovaných datasetů. Pro rozdělené datasety je čtení pro kompresi u řádků 7krát pomalejší, u sloupců 3krát.

Čtení řádku						
	Složený dataset			Rozdělený dataset		
Počet bodů	Bez kom.	2. úroveň	9. úroveň	Bez kom.	2. úroveň	8. úroveň
10000	1, 96ms	346, 42ms	314, 61ms	31, 21ms	297, 47ms	268, 98ms
20000	3, 49ms	690, 03ms	629, 89ms	72, 58ms	589, 05ms	533, 09ms
50000	10, 64ms	1719, 88ms	-	196, 39ms	1420, 64ms	1297, 67ms
Čtení sloupce						
	Složený dataset			Rozdělený dataset		
Počet bodů	Bez kom.	2. úroveň	9. úroveň	Bez kom.	2. úroveň	8. úroveň
10000	22, 93ms	344, 86ms	317, 42ms	115, 96ms	339, 85ms	320, 02ms
20000	38, 92ms	690, 30ms	635, 46ms	224, 86ms	693, 78ms	636, 70ms
50000	105, 11ms	1701, 50ms	-	551, 14ms	1623, 74ms	1526, 66ms

Tabulka 20: Čtení řádků a sloupců struktury Traffic pro různé úrovně komprese

Výsledky testů jedné proměnné budou opět z důvodu rozsáhlosti testů uvedeny jen v příloze C. Během těchto testů se opět projevilo, že jsou rozdělené datasety pro čtení jedné proměnné z komprimovaných datasetů výrazně lepší než datasety složené. Také se znovu ukázalo, že rozdíly mezi jednotlivými kompresemi jsou u čtení zanedbatelné.

6.6 Testování rychlosti čtení náhodných bodů dat

Třetím a posledním testem čtení byl test čtení náhodných bodů v datasetu. Během tohoto testu se provádělo čtení z datasetů dvou velikostí, 10000×10000 a 50000×50000. Pro testy byly použity dvě funkce, funkce *readPoints* pro čtení náhodných bodů celé struktury struktury a funkce *readPointsParam* pro čtení náhodných bodů jedné proměnné struktury. Zvolené proměnné pro tento test byly opět DEGCEL a PC. Pro každý test se nejdříve vygenerovalo 10000, 20000 nebo 50000 náhodných souřadnic z celého datasetu a následně se změřilo jak dlouho trvá přečtení bodů na těchto souřadnicích.

6.6.1 Výsledky pro různé chunky

Výsledky testu čtení náhodných bodů pro strukturu Weather můžeme vidět v tabulce 21. Pro strukturu Traffic jsou tyto výsledky v tabulce 22. Během testu dosáhlo ve všech případech nejlepších výsledků chunkování 100×100. Rozdíly mezi jednotlivými chunkováními, ale nebyly nijak zásadní. Dá se tedy usuzovat, že má velikost chunků na čtení náhodných bodů jen malý vliv, což je vcelku logické, když si uvědomíme, že při čtení náhodného bodu téměř pokaždé přistupujeme k jinému chunku, než při čtení předchozího bodu. Tím se také může vysvětlit, proč nejlepších výsledků dosáhlo chunkování s největšími rozměry. Je totiž pravděpodobné, že při čtení z takového datasetu, bylo nejvíce přístupů do stejných chunků.

	Dataset 10000×10000		Dataset 50000×50000	
	Složené d.	Rozdělené d.	Složené d.	Rozdělené d.
Počet bodů	100×100	100×100	100×100	100×100
10000	2, 42s	2, 73s	176, 98s	696, 08s
20000	3, 38s	4, 75s	475, 17s	1353, 58s
50000	14, 28s	19, 23s	801, 57s	2909, 36s

Tabulka 21: Čtení náhodných bodů struktury Weather pro různé velikosti chunků

	Dataset 10000×10000		Dataset 50000×50000	
	Složené d.	Rozdělené d.	Složené d.	Rozdělené d.
Počet bodů	50×50	100×100	100×100	100×100
10000	11, 55s	142, 51s	122, 46s	1791, 94s
20000	63, 22s	147, 00s	215, 04s	3141, 00s
50000	83, 41s	362, 36s	529, 39s	8099, 01s

Tabulka 22: Čtení náhodných bodů struktury Traffic pro různé velikosti chunků

Z tabulek také můžeme zjistit, že obrovský vliv na rychlost čtení náhodných bodů, má velikost datasetu, ze kterého čteme. V případě čtení náhodných bodů z větších datasetů o rozměrech 50000×50000 jsou tyto časy opravdu obrovské a téměř dosahují hodnot pro čtení celých bloků

těchto datasetů. V souvislosti s tímto, by mohlo být zajímavé otestovat čtení náhodných, ale seřazených bodů. Je možné, že v takovém případě by bylo mnohem více přístupů do stejných chunků a čas čtení by se tak výrazně zkrátil. Poslední informací, které si v tabulkách můžeme všimnout je to, že rozdělené datasety jsou pomalejší než datasety složené. V případě větších datasetů 50000 × 50000 se tento rozdíl ještě zvyšuje.

V tabulkách 23 a 24 můžeme vidět výsledky pro čtení náhodných bodů jedné proměnné struktury. I u tohoto testu byl největší rozdíl v tom, jestli čteme náhodné body z datasetu o velikosti 10000 × 10000 nebo 50000 × 50000. Znovu se také ukázalo, že lepších časů při čtení jedné proměnné dosahují datasety rozdělené. Nejlepší časy pro složené datasety mělo většinou chunkování 50 × 50, pro rozdělené 100 × 100. Rozdíly mezi jednotlivými velikostmi chunků byly opět malé. Můžeme si také všimnout, že v některých případech bylo čtení více bodů rychlejší než čtení méně bodů. Toto bude pravděpodobně způsobeno náhodnou podstatou testů.

	Dataset 10000 × 10000		Dataset 50000 × 50000	
	Složené d.	Rozdělené d.	Složené d.	Rozdělené d.
Počet bodů	50 × 50	100 × 100	50 × 50	100 × 100
10000	5, 46s	0, 57s	193, 60s	100, 54s
20000	3, 90s	0, 54s	304, 90s	234, 37s
50000	4, 04s	1, 03s	607, 28s	412, 23s

Tabulka 23: Čtení náhodných bodů jedné proměnné struktury Weather pro různé velikosti chunků

	Dataset 10000 × 10000		Dataset 50000 × 50000	
	Složené d.	Rozdělené d.	Složené d.	Rozdělené d.
Počet bodů	50 × 50	100 × 100	100 × 100	100 × 100
10000	21, 54s	0, 36s	114, 69s	102, 68s
20000	6, 13s	0, 38s	255, 89s	153, 82s
50000	6, 84s	0, 65s	549, 37s	238, 17s

Tabulka 24: Čtení náhodných bodů jedné proměnné struktury Traffic pro různé velikosti chunků

6.6.2 Výsledky pro kompresi

Zajímavé výsledky přinesl test čtení náhodných bodů pro komprimované datasety, které nalezneme pro strukturu Weather v tabulce 25 a pro strukturu Traffic v tabulce 26. Při porovnání časů čtení z komprimovaných a nekomprimovaných datasetů totiž zjistíme, že komprimované datasety jsou ve většině případů rychlejší než nekomprimované. Tento výsledek se dá nejlépe vysvětlit tím, že v HDF5 je vyhledávání bodů pomalejší než samotná dekomprese, a proto je rychlejší čtení z komprimovaného, tedy menšího datasetu, ve kterém se tyto body vyhledávají rychleji.

Dataset 10000 × 10000				
	Složené datasety		Rozdělené datasety	
Počet bodů	Bez komprese	2.úroveň	Bez komprese	2.úroveň
10000	2, 42s	26, 48s	2, 73s	21, 80s
20000	3, 38s	49, 75s	4, 75s	45, 02s
50000	14, 28s	120, 23s	19, 23s	95, 99s
Dataset 50000 × 50000				
	Složené datasety		Rozdělené datasety	
Počet bodů	Bez komprese	2.úroveň	Bez komprese	2.úroveň
10000	176, 98s	153, 07s	696, 08s	527, 47s
20000	475, 17s	334, 06s	1353, 58s	805, 52s
50000	801, 57s	611, 03s	2909, 36s	1385, 88s

Tabulka 25: Čtení náhodných bodů struktury Weather pro různé úrovně komprese

Další zajímavé výsledky u tohoto testu nejsou, opět jsou jen velmi nevýrazné rozdíly mezi jednotlivými kompresemi, složené datasety jsou rychlejší než rozdělené a velký vliv na čas čtení má celková velikost datasetu.

Dataset 10000 × 10000				
	Složené datasety		Rozdělené datasety	
Počet bodů	Bez komprese	2.úroveň	Bez komprese	2.úroveň
10000	96, 64s	35, 25s	142, 51s	28, 62s
20000	148, 54s	66, 20s	147, 00s	53, 94s
50000	325, 17s	160, 30s	362, 36s	130, 27s
Dataset 50000 × 50000				
	Složené datasety		Rozdělené datasety	
Počet bodů	Bez komprese	2.úroveň	Bez komprese	2.úroveň
10000	122, 46s	134, 17s	1791, 94s	578, 07s
20000	215, 04s	261, 11s	3141, 00s	751, 84s
50000	592, 39s	509, 18s	8099, 01s	1201, 93s

Tabulka 26: Čtení náhodných bodů struktury Traffic pro různé úrovně komprese

Výsledky testů čtení jedné proměnné z komprimovaného datasetu, zde opět nebudou uvedeny, můžeme je ale nalézt v příloze C. V tomto testu se znovu objevily úkazy z testů předešlých. Opět se ukázal pozitivní vliv komprese na čtení náhodných bodů, rovněž rozdíly jednotlivých kompresí byly minimální. Lepších výsledků tradičně dosáhly rozdělené datasety, které se zdají být lepší ve všech testech pro kompresi a čtení jedné proměnné.

6.7 Zhodnocení dosavadních testů

Vzhledem k veliké rozsáhlosti testů zde bude uvedeno krátké zhodnocení testů, převážně z hlediska volby ideální velikosti chunků a komprese. Také zde budou uvedena jistá doporučení pro užívání složených a rozdělených datasetů.

K zorientování se, které velikosti chunků uspěly v kterém testu nám pomůžou tabulky 27 a 28. Tyto tabulky obsahují umístění jednotlivých velikostí chunků v konkrétních testech. V tabulce 27 můžeme vidět jak dopadly tyto chunkování pro strukturu Weather.

	Složené datasety				
Test	100×100	50×50	10×10	100×1	1×100
Zápis	4.	3.	2.	1.	5.
Blok	4.	2.	3.	1.	5.
Blok pr.	4.	2.	3.	1.	5.
Řádek	4.	2.	3.	1.	5.
Řádek pr.	4.	2.	3.	1.	5.
Sloupec	3.	2.	4.	5.	1.
Sloupec pr.	3.	2.	4.	5.	1.
Body	2.	1.	3.	4.	5.
Body pr.	3.	1.	2.	4.	5.
	Rozdělené datasety				
Test	100×100	50×50	10×10	100×1	1×100
Zápis	2.	3.	4.	1.	5.
Blok	3.	2.	4.	1.	5.
Blok pr.	2.	3.	4.	1.	5.
Řádek	3.	2.	4.	1.	5.
Řádek pr.	3.	2.	4.	1.	5.
Sloupec	1.	3.	4.	5.	2.
Sloupec pr.	2.	3.	4.	5.	1.
Body	1.	2.	3.	4.	5.
Body pr.	1.	2.	5.	3.	4.

Tabulka 27: Umístění jednotlivých chunkování v testech - struktura Weather

Z tabulky je jasně patrné, že chunkování 100×1 bylo nejlepší v testech zápisu, čtení bloků dat a čtení řádků. Naopak v testech čtení sloupců a náhodných bodů bylo toto chunkování jedno z nejhorších. Pro složené datasety se stabilně mezi nejlepšími drželo chunkování 50×50, které se tak dá použít jako univerzální pro složené datasety a strukturu Weather. U rozdělených datasetů už to tak jednoznačné nebylo, chunkování 100×100 a 50×50 byly přibližně rovnocenné. Chunkování 100×100 bylo lepší v zápise, čtení sloupců a čtení náhodných bodů. Chunkování 50×50 bylo zase lepší pro čtení bloků a čtení řádků.

Výběr správného chunkování tak záleží hlavně na typech operací, které nad datasetem budeme provádět. Pokud nevyužijeme čtení sloupců ani čtení náhodných bodů, tak je nejlepší

použít chunky o velikostech 100×1. V případě, že tyto operace chceme využívat, tak se zdá lepší použít pro složené datasety chunkování 50×50 a pro rozdělené buď 100×100 nebo 50×50. Chunkování 1×100 lze využít snad jen v případě, že budeme přistupovat k datům jenom po sloupcích a zároveň se data nebudou v průběhu měnit a přepisovat. I v takovém případě je ale nutné zvážit jestli nezvolit raději jedno z univerzálnějších chunkování.

V tabulce 28 můžeme vidět srovnání jednotlivých chunkování pro strukturu Traffic. I u této struktury se ukázalo, že chunkování 100×1 je nejlepší variantou pro zápis, čtení bloků dat a čtení řádků, zatímco čtení pro čtení sloupců a náhodných bodů není toto chunkování příliš vhodné. Pokud bychom chtěli vybrat chunkování, které má rozumné výsledky pro všechno, tak bychom pravděpodobně vybrali chunkování 100×100, které se většinou drželo mezi dvěma prvními. Ale i toto chunkování má jeden velký nedostatek. Při čtení bloku dat a pro složené datasety i při čtení bloku jedné proměnné struktury mělo toto chunkování velmi špatné výsledky. Tyto špatné výsledky se však projevily jen u varianty čtení po řádcích a můžeme tedy využít variantu čtení po sloupcích.

	Složené datasety				
Test	100×100	50×50	10×10	100×1	1×100
Zápis	2.	4.	3.	1.	5.
Blok	5.	3.	2.	1.	4.
Blok pr.	5.	3.	2.	1.	4.
Řádek	2.	4.	3.	1.	5.
Řádek pr.	2.	4.	3.	1.	5.
Sloupec	2.	3.	4.	5.	1.
Sloupec pr.	2.	3.	4.	5.	1.
Body	1.	2.	4.	5.	3.
Body pr.	1.	2.	5.	4.	3.
	Rozdělené datasety				
Test	100×100	50×50	10×10	100×1	1×100
Zápis	2.	3.	4.	1.	5.
Blok	4.	2.	3.	1.	5.
Blok pr.	2.	3.	4.	1.	5.
Řádek	2.	3.	4.	1.	5.
Řádek pr.	2.	3.	4.	1.	5.
Sloupec	2.	3.	4.	5.	1.
Sloupec pr.	1.	3.	4.	5.	2.
Body	1.	3.	2.	4.	-
Body pr.	1.	4.	2.	3.	-

Tabulka 28: Umístění jednotlivých chunkování v testech - struktura Traffic

I u struktury Traffic bude volba nejlepšího chunkování záviset na typech operací, které budeme používat. Opět platí, že jestliže nebudeme používat sloupců řádků a náhodných bodů, tak

je nejlepší použít chunkování 100×1. Pokud je využívat budeme, tak je pravděpodobně nejlepší použít chunkování 100×100 a to i přes jeho slabinu u čtení bloku dat po řádcích.

Z testů komprimovaných datasetů jsme zjistili spoustu zajímavých výsledků. Při testu velikostí jsme zjistili, že vyšší úrovně komprese nabízejí jen velmi malé zvýšení kompresního poměru. Také jsme zjistili, že jednotlivé komprese bývají přibližně stejně rychlé pro většinu testů čtení, v některých případech bylo dokonce čtení z datasetů s vyšší úrovní komprese rychlejší než z datasetů s nižší úrovní. Bohužel je toto malé zlepšení ve velikostech a rychlostech čtení vyváženo mnohem delším zápisem. Tato nevýhoda, ale nemusí být podstatná pokud budeme data zapisovat postupně s časovými intervaly. Takový zápis je zároveň velmi častý pro časové řady, pro které je tak tato nevýhoda zanedbatelná. Velmi zajímavé zjištění nám přinesl test čtení náhodných bodů. U tohoto testu dopadly komprimované datasety lépe než datasety nekomprimované. Toto nám nabízí možnost využít kompresi k zvýšení výkonu, pokud bude preferovanou operací nad daty čtení náhodných bodů.

Zhodnocením testů pro složené a rozdělené datasety můžeme dospět k závěru, že složené datasety jsou lepší pro nekomprimované datasety, pokud čteme data celé struktury. Pro čtení jedné proměnné jsou pak lepší datasety rozdělené, které jsou lepší i pro všechny operace s komprimovanými datasety.

6.8 Testování paralelního přístupu k HDF5 souboru

Posledním testem, který v rámci této práce proběhl byl test paralelního čtení více procesů z jednoho souboru. Během testu četlo více procesů z jednoho složeného, nekomprimovaného datasetu o velikosti 50000×50000 a velikostech chunků 100×100. Každý proces četl pomocí funkce *readBlock* blok dat o rozměrech 2000×2000. Počty procesů pro, které tento test proběhl, byly 1 (1 jádro), 24 (1 uzel), 48 (2 uzly) a120 (5 uzlů). Také byly otestovány dvě varianty testu, v první variantě četly procesy vždy stejný blok dat, v druhé četly procesy každý svůj vlastní blok dat. Za výsledek testu byl považován nejpomalejší čas ze všech procesů.

Výsledné časy pro obě struktury a oba typy testů můžeme vidět v tabulce 29. Výsledky u struktury Weather naznačují, že počet paralelních přístupů k souboru má vliv na rychlost čtení, ale tento vliv není nijak velký. Větší vliv se ukazuje spíše při porovnání čtení různých a stejných bloků, pokud čteme různé bloky, tak je čas čtení pomalejší, než při čtení bloků stejných.

Výsledky pro strukturu Traffic jsou docela překvapivé, u čtení stejného bloku se při navýšení počtu procesů dokonce v některých případech snižoval čas čtení. Naopak u čtení různých bloků vidíme jasně stoupající časy, což odpovídá struktuře Weather. I zde ale nalezneme anomálii u přístupu ze 120 procesů, kde se čas jednotlivých čtení výrazně zvýšil oproti předcházejícím měřením. Tento výrazný skok může být způsoben větší velikostí struktury Traffic a tudíž také většímu objemu dat, které 120 procesů ze souboru vyžaduje.

Závěrem tohoto testu je, že HDF5 dobře zvládá paralelní čtení více procesů a nárůst v čase s navýšením počtu procesů je minimální.

Počet přistupujících procesů	Weather		Traffic	
	Stejný blok	Různé bloky	Stejný blok	Různé bloky
1 (1 jádro)	8, 96s	8, 96s	14, 73s	14, 73s
24 (1 uzel)	8, 60s	19, 29s	5, 74s	29, 57s
48 (2 uzly)	10, 51s	18, 17s	3, 14s	38, 20s
120 (5 uzlů)	11, 10s	21, 84s	4, 17s	360, 43s

Tabulka 29: Test paralelního přístupu více procesů

7 Závěr

Cílem této práce bylo prozkoumat různé přístupy, které jsou v dnešní době používány k ukládání velkých objemů dat, reprezentovaných časovými řadami a analyzovat jejich využití pro HPC clustery. V rámci práce byly popsány vědecké datové formáty HDF5, CDF a NetCDF a také databáze NoSQL a knihovna Adios. Také zde byl popsán souborový systém Lustre a jeho přínos pro paralelní přístup k datům uloženým na HPC clusterech.

Dalším cílem bylo navrhnout a implementovat nad HDF5 knihovnou nadstavbu pro práci s časovými řadami, popsat vytvořené API a vytvořit dokumentaci tak, aby se nad touto nadstavbou daly implementovat různé adaptéry pro její využití v technologiích jako jsou C# nebo Excel. Posledním cílem této práce bylo otestovat funkčnost knihovny na testovací kolekci dat. Všechny cíle práce byly úspěšně splněny. Během testů byly také zjištěny podstatné informace, které můžou být v budoucnu využity pro optimalizaci ukládání dat pomocí HDF5.

Na tuto práci je možné v mnoha směrech navázat. Místo HDF5 je možné použít jiný z přístupů, které byly v rámci této práce prozkoumány. Pro HPC prostředí se zvlášť zajímavým přístupem jeví využití knihovny Adios. Další z možností, kterou je možno se dále ubírat je využití souborového systému Lustre společně s HDF5 a ověřit, jestli jejich spojení zvýší výkon při přístupu k datům. Ani u samotného HDF5 nejsou všechny možnosti vyčerpané, je zde prostor pro experimenty s velikostí chunk cache, nebo HDF5 ovladači. V nové verzi HDF5 také přibude možnost paralelního čtení a zápisu. Ani samotná knihovna vytvořená v rámci této práce nedosáhla hranice svých možností. Je zde prostor pro paralelizaci knihovny, nebo rozšíření knihovny pro ukládání jiných než časových dat.

Literatura

[1] *Adios - Oak Ridge National Laboratory* [online]. Dostupné z: https://www.olcf.ornl.gov/center-projects/adios/ [citováno 12.dubna 2016].

[2] HEIJMANS, Jan. *An Introduction to Distributed Visualization*. Delft University of Technology. Faculty of Information Technology and Systems. February 2002.

[3] *Big data - Wikipedia* [online]. Dostupné z: https://en.wikipedia.org/wiki/Big_data [citováno 23.března 2016].

[4] *CDF User's Guide - Goddard Space Flight Center* [online]. Dostupné z: http://cdaweb.gsfc.nasa.gov/pub/software/cdf/doc/cdf361/cdf361ug.pdf [citováno 4.dubna 2016].

[5] *Common Data Format - Goddard Space Flight Center* [online]. Dostupné z: http://cdf.gsfc.nasa.gov/ [citováno 4.dubna 2016].

[6] *Compute Nodes - IT4Innovations Docs* [online]. Dostupné z: https://docs.it4i.cz/salomon/compute-nodes [citováno 22.dubna 2016].

[7] *Database variants explained : SQL or NoSQL? Is that really the question?* [online]. Dostupné z: https://kvaes.wordpress.com/2015/01/21/database-variants-explained-sql-or-nosql-is-that-really-the-question/ [citováno 14.dubna 2016].

[8] *Deflate - Wikipedia* [online]. Dostupné z: https://en.wikipedia.org/wiki/DEFLATE [citováno 20.dubna 2016].

[9] *Goddard Space Flight Center - NASA* [online]. Dostupné z: https://www.nasa.gov/goddard [citováno 4.dubna 2016].

[10] *Golomb coding - Wikipedia* [online]. Dostupné z: https://en.wikipedia.org/wiki/Golomb_coding [citováno 20.dubna 2016].

[11] *HDF5 Software Documentation - The HDF Group* [online]. Dostupné z: https://www.hdfgroup.org/HDF5/doc/index.html [citováno 17.dubna 2016].

[12] *HDF5 User's Guide* [online]. Dostupné z: https://www.hdfgroup.org/HDF5/doc/UG/-HDF5_Users_Guide-Responsive%20HTML5 [citováno 20.dubna 2016].

[13] *HERE* [online]. Dostupné z: https://company.here.com/here/ [citováno 23.dubna 2016].

[14] *Hierarchical Data Format - Wikipedia* [online]. Dostupné z: https://en.wikipedia.org/wiki/Hierarchical_Data_Format [citováno 27.března 2016].

[15] ZUBEK, Petr. *Implementace adaptérů pro přístup k datům reprezentovaným časovými řadami*. Ostrava, 2016. Vysoká Škola Báňská - Technická Univerzita Ostrava. Fakulta elektrotechniky a informatiky. Vedoucí diplomové práce Ing. Kateřina Slaninová, Ph.D.

[16] PADHY, Rabi Prasad. RDBMS to NoSQL: Reviewing Some Next-Generation Non-Relational Database's. *International Journal of Advanced Engineering Sciences and Technologies*, 2011, Vol No. 11, Issue No. 1, 015 - 030.

[17] *Introduction to Parallel I/O, Lustre, & ADIOS* [online]. Dostupné z: https://www.erdc.hpc.mil/docs/Tips/garnet-lustre-adios.pdf [citováno 12.dubna 2016].

[18] *Lustre* [online]. Dostupné z: http://lustre.org/ [citováno 15.dubna 2016].

[19] *National Center for Supercomputing Applications* [online]. Dostupné z: http://www.ncsa.illinois.edu/ [citováno 27.března 2016].

[20] *NetCDF - Wikipedia* [online]. Dostupné z: https://en.wikipedia.org/wiki/NetCDF [citováno 7.dubna 2016].

[21] *Network Common Data Form (NetCDF) - UCAR* [online]. Dostupné z: http://www.unidata.ucar.edu/software/netcdf/ [citováno 7.dubna 2016].

[22] *NoSQL - Your Ultimate Guide to the Non-Relational Universe!* [online]. Dostupné z: http://nosql-database.org/ [citováno 10.dubna 2016].

[23] HADJIGEORGIOU, Christoforos. *RDBMS vs NoSQL: Performance and Scaling Comparison*. MSc in High Performance Computing. The University of Edinburgh. August 2013.

[24] *Scientific Data Formats - IDL Manual* [online]. Dostupné z: http://www.geo.mtu.edu/-geoschem/docs/IDL_Manuals/SCIENTIFIC%20DATA%20FORMATS.pdf [citováno 24.března 2016].

[25] *Storage - IT4Innovations Docs* [online]. Dostupné z: https://docs.it4i.cz/salomon/storage [citováno 22.dubna 2016].

[26] *The gzip home page* [online]. Dostupné z: http://www.gzip.org/ [citováno 20.dubna 2016].

[27] *The HDF Group* [online]. Dostupné z: https://www.hdfgroup.org [citováno 27.března 2016].

[28] *The HDF levels of interaction - The HDF Group* [online]. Dostupné z: https://www.hdfgroup.org/products/hdf4/whatishdf.html [citováno 28.března 2016].

[29] *University Corporation for Atmospheric Research* [online]. Dostupné z: http://www2.ucar.edu/ [citováno 7.dubna 2016].

[30] *Why HDF - The HDF Group* [online]. Dostupné z: https://www.hdfgroup.org/why_hdf/ [citováno 27.března 2016].

A Zdrojové kódy knihovny pro ukládání časových řad do formátu HDF5

Příloha na CD/DVD:

- AttributeCreator.cpp
- AttributeCreator.h
- AttributeReader.cpp
- AttributeReader.h
- AttributeWritter.cpp
- AttributeWritter.h
- CompoundDatasetCreator.cpp
- CompoundDatasetCreator.h
- CompoundDatasetReader.cpp
- CompoundDatasetReader.h
- CompoundDatasetWritter.cpp
- CompoundDatasetWritter.h
- DividedDatasetCreator.cpp
- DividedDatasetCreator.h
- DividedDatasetReader.cpp
- DividedDatasetReader.h
- DividedDatasetWritter.cpp
- DividedDatasetWritter.h
- FileCreator.cpp
- FileCreator.h
- GroupCreator.cpp
- GroupCreator.h
- Info.cpp
- Info.h
- InfoFunctions.cpp
- InfoFunctions.h
- IO.cpp
- IO.h
- ItemDeleter.cpp
- ItemDeleter.h
- main.cpp
- stdafx.h
- StringParser.cpp
- StringParser.h
- StructInfo.cpp

StructInfo.h
TrafficInfo.cpp
TrafficInfo.h
WeatherInfo.cpp
WeatherInfo.h

B Dokumentace knihovny

Příloha na CD/DVD:

Dokumentace.pdf

C Kompletní výsledky testů

Příloha na CD/DVD:

Weather.xlsl

Traffic.xlsl